Parallel Query Processing on the Grid

Anastasios GOUNARIS [a], Alvaro A.A. FERNANDES [b],
Apostolos N. PAPADOPOULOS [a], and Christos YFOULIS [c]
[a] *Aristotle University of Thessaloniki, Greece*
[b] *University of Manchester, United Kingdom*
[c] *Technological Educational Institute of Thessaloniki, Greece*

Abstract. Database queries offer an easy-to-use declarative manner for describing complex data management tasks. Query processing technologies have been evolving for decades; however the emergence of the Grid creates a new setting in which novel research issues and challenges have arisen. This chapter discusses how Grid-oriented and/or service-based query processors differ from traditional ones, and focuses on three complementary research issues, namely, how to schedule parallel database queries over non-dedicated, distributed resources; how to mitigate the impact of increased data transfer cost; and how to perform load balancing in this new setting. In addition, we discuss how parallel spatio-temporal query processing techniques can be applied to a Grid environment. The discussion revolves around the development of the OGSA-DQP system, which is a pioneer open-source service-based query processing system that enables parallel query execution over Grid resources, and the way some of the most prominent issues about its performance were addressed. The unique characteristics of the scheduling problem of arbitrarily parallel queries over heterogeneous resources have motivated the development of a new hill-climbing algorithm. For the problems of increased data transmission cost and load balancing, due to the highly volatile conditions, techniques founded on control theory are examined. The emphasis of this chapter is on both the description of a real Grid-enabled parallel query processor and the presentation of the different approaches to tackling each of the afore-mentioned problems including the limitations of the current state-of-the-art solutions.

Keywords. distributed query processing, adaptive query processing, query scheduling, load balancing, OGSA-DQP, grid computing, control systems, hill climbing

Introduction

Parallel query processing is a mature technology aiming at providing high performance, high availability and scalability in database systems. The key concept is to allow database management systems (DBMSs) to benefit from the use of multiple resources, such as CPU cores, by executing (sub-)tasks concurrently. By pooling more resources to process intensive queries, these queries can, in principle, run faster, while mitigating, in the majority of the cases, the impact of a single node's failure or unavailability. Intensive queries are queries that process large volumes of data or include complex and expensive predicates or the combination of both. To date, parallel database systems are widely spread and almost all commercial systems come with a parallel flavor, capitalizing on the fact that parallel query processing is nowadays well understood.

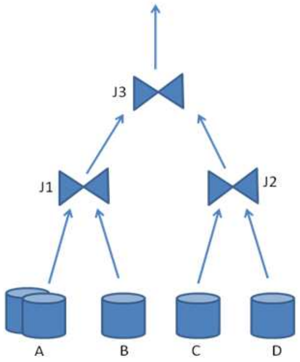


Figure 1. Example of a 3-join query plan that retrieves data from 4 logical data sources.

Parallelism in database queries can be applied either at the inter-query level or at the intra-query level, where the same query is parallelized across multiple resources. Inter-query parallelism is very common in either centralized or distributed systems and involves the concurrent execution of multiple queries with effective and efficient query state sharing being one of the biggest persisting challenges [9]. In this chapter, we focus on intra-query parallelism, which can be further divided into independent parallelism, pipelined parallelism and partitioned parallelism. Typically, a query plan is represented as a directed graph, in which the nodes correspond to algebraic operators and edges denote data flow. Independent parallelism occurs when two or more independent subgraphs of the query graph corresponding to the user query are being executed concurrently. In pipelined –or inter-operator parallelism–, two or more connected graph fragments are executed in parallel, whereas, partitioned –or intra-operator– parallelism refers to the case where a node in the query graph is cloned into several instances by partitioning the input data and allowing each clone to operate on a different data partition. An example is shown in Figure 1. Assuming that all joins in the figure are implemented as pipelined hash joins, all three forms of intra-query parallelism can occur in this query plan: J1 and J2 can be processed independently, J1 and J3 can be processed in a pipelined fashion, and data from logical source A can be retrieved in a partitioned manner. Obviously, these forms of parallelism can be applied to multi-threaded single processor systems as well; however, in that case, they yield limited benefits.

The architectural choices that naturally support parallel query processing can be classified in four broader categories: (i) shared nothing architectures, according to which the processing elements are interconnected with some sort of network and are independent from each other in the sense that they have exclusive access to their own primary and secondary memory; (ii) shared memory architectures, which differ from the previous in that the primary memory is shared across the processing elements; (iii) shared disk architectures, in which processors share only secondary memory devices while keeping full autonomy over their primary memory; and (iv) shared everything architectures, which refer to tightly coupled settings where both primary and secondary memory is common to all processors. Parallel grid computing heavily relies on the first category [48,6], since non-compromised autonomy of individual nodes and communication between various nodes using standardized network interfaces are key aspects of the Grids. Nevertheless, a main difference is that typically, in parallel shared nothing systems the nodes are homogeneous, whereas, in realistic Grids, the nodes belong to clusters that have different characteristics, also in terms of access rights and local policies to be obeyed, and the communication latency is non-negligible.

Parallel query processing on the Grid is also related to wide area query processing and semantic data integration [25], which is out of the scope of this chapter. Additionally, in distributed environments, there are two main architectural alternatives, namely client-server systems and peer-to-peer ones. Peer-to-peer is the most general architecture allowing every site to act both as a server that stores parts of the database and as a client that executes application programs and initiates queries [29]. In classical distributed query processing, some parts of the queries are executed at a central place, whereas other parts, in the form of subqueries, are executed, after some translation steps, in the component database engines. According to this model, both inter- and intra-operator parallelism are limited. The latter occurs only in the case where a query retrieves data from a database view that is physically partitioned across multiple nodes.

Grid query processing generalizes the afore-mentioned model and allows arbitrary sites to participate in the execution; query engines can be spawned at nodes in a way that permits individual operators to be mapped to Grid computational resources and places no limit on the degree of intra-operator parallelism for any operator in the query plan. This flexibility comes at the expense of higher complexity in the query processing strategies and in the system architecture. In order for individual data sources to be capable of dynamically joining a federation and computational resources to be capable of dynamically join the resource pool and execute query fragments, common interfaces and mechanisms must be devised. A generic solution that has been adopted by systems such as OGSA-DQP (Open Grid Service Architecture - Distributed Query Processing) [1,34], is to expose both data sources and execution engines as Web Services with well defined interfaces and standard access methodologies. However, these methodologies are more usually based on exchanges of verbose messages resulting in significant communication cost.

Consequently, an important challenge is to devise methods that tackle the issue of increased cost of data transmission. Additional open issues with respect to query optimization and execution include the need for novel algorithms for deciding the degree of parallelism and the selection of remote machines to be employed, which is a problem that does not arise in parallel homogeneous settings, and for balancing the load across machines under highly volatile conditions. Complementarily to these, more efficient solutions for placing and replicating the data must be developed. Other research challenges stem from the fact that, in a Grid environment, the characteristics of the machines vary and are subject to frequent changes calling for adaptive query processing techniques [9]. Also, the optimization criteria may be different from the traditional ones such as query response time, some peers may be unreliable, and queries may be very expensive and thus sharing query state across multiple queries becomes more important. It is worthy to note that solving all these issues is of more generic interest, since other areas, such as dataspaces [13,26], can benefit from such solutions.

This chapter aims to investigate research issues and directions for future work related to parallel query processing in Grid environments by presenting possible solutions and discussing the extent to which these solutions have addressed such issues. The solutions have been developed in close relationship with the OGSA-DQP system [1,34], which is a pioneer service-based query processing system that enables parallel query execution over Grid resources and, indirectly, they reflect the early experiences with a real Grid query processing prototype and its limitations. More specifically, in this chapter, first we present the OGSA-DQP system (Section 1). Next, we deal with three of the research issues mentioned before, namely techniques for reducing the data transfer cost in service-based environments (Section 2), scheduling algorithms tailored to Grid query processing (Section 3), and advanced load balancing solutions (Section 4). The need for parallel query processing on the Grid is more felt in intensive queries, such as those expected to occur in many scientific scenarios, which may deviate to a certain extent from textbook SQL ones (e.g., spatio-temporal queries). In Section 5, we present how advanced spatio-temporal techniques can be transferred to a Grid setting. The chapter conclusions are in Section 6.

1. Service-based query processors: the OGSA-DQP case

This section describes one implemented approach to service-based distributed query processing, viz., that embodied by OGSA-DQP [34][1].

1.1. Motivation

Service-oriented architectures were conceived along with the idea of service orchestration as the concrete specification mechanism to embody process models. Thus, if businesses expose their business processes as services, one could envisage a virtual organization emerging from the orchestration of services provided by different concrete organizations.

The most popular approach to service orchestration is based on workflows. Service-based workflow languages have been developed that allow high-level service orchestration. Much effort has been devoted to making service orchestration simple. However, because workflow languages are imperative, the risk is always present that expensive expertise is required to ensure that a workflow will be effective and efficient (i.e., that it will, respectively, satisfy the functional and non-functional requirements placed upon it).

Arguably, the main motivation for deploying the distributed query processing (DQP) paradigm in service-oriented architectures is the fact that it allows an important class of concrete, effective and efficient service orchestrations to be derived from declarative specifications (e.g., high-level SQL-like queries) in automated fashion by means of a compiler/optimizer.

Given that many (if not most) service orchestrations will involve data resources virtualized as data services, it follows that many service orchestrations will be characterizable as computations that can be reduced to retrieval, transformation and transport operations applied to flows of data grounded on data services.

OGSA-DQP has indeed championed an approach that has been termed *declarative service orchestration* and has been guided by an insight that there is mutual benefit to be reaped by DQP from its deployment in Grid settings, as well as by the Grid vision for distributed computing from being seen, where appropriate, as a generalization of distributed data management platforms. Thus, one can argue that DQP benefits from Grids by building upon the sophisticated mechanisms for securely and transparently discover-

[1] This section is closely based on the account provided in [34]. For an earlier account of OGSA-DQP, see [1].

ing and harnessing heterogeneous data and computational resources, whereas Grids benefit from DQP because the latter endows the former with a declarative interface that is well-known to software engineers and well-known to admit of reliable, robust, efficient, scalable implementations.

1.2. Approach

OGSA-DQP is service-based in two distinct senses. Firstly, externally, OGSA-DQP is exposed as a data service (i.e., it exposes DQP capabilities as service invocations). In this way, invoking OGSA-DQP capabilities is tantamount to having access to a full-blown federated database whose components are data services, i.e., virtualized data resources. Secondly, internally, OGSA-DQP capabilities are themselves implemented as service orchestrations.

In this way, when OGSA-DQP receives a request to evaluate a distributed query, it responds to that request by harnessing the services that front-end the computational resources required to evaluate the distributed query execution plan derived for the requested query. Thus, in any distributed query execution plan produced by OGSA-DQP, the non-leaf nodes, i.e., the distributed plan fragments, are assigned to evaluator services harnessed from the collection of available computational resources, and the leaves are assigned to the virtualized data resources required to compute the answer to the query.

1.3. Usage scenarios

The SQL-based OGSA-DQP query language supports queries that (in their `FROM` clause) make reference to data services (i.e., remote, autonomously-owned, virtualized data resources such as, in bioinformatics, a protein sequence database) and to analysis services, such as, again in bioinformatics, a virtualized computational resource that supports BLAST (Basic Local Alignment Search Tool) sequence analysis services. In addition, OGSA-DQP relies on another specific type of virtualized computational resource, viz., evaluator services, to which the responsibility for evaluating a plan fragment can be assigned.

OGSA-DQP has been designed and implemented to occupy a region in the space of solutions for the problem of using the Grid for DQP that is characterized by two major properties. Firstly, OGSA-DQP supports data integration at the low-entry-cost end of the spectrum. In this sense, OGSA-DQP follows the Grid vision of agile assembly of virtualized resources to form coherent but short-lived configurations thereof, in support of applications whose lifecycle is far too compressed to justify the effort required in setting up a meticulously-designed global schema. Thus, OGSA-DQP does not provide facilities for schema matching and schema mapping that are required when such a specifically-intended, long-lasting global schema needs to be exposed by the federated database. Instead, OGSA-DQP provides facilities for importing the union of participating schemas without precluding that, downstream from the delivery of query results, schema mapping services (in which such queries would be seen, e.g., as views) are additionally provided. In this respect, OGSA-DQP is therefore most useful when deployed as middleware for agile, efficient, expressive declarative access to multiple provider data services. Secondly, OGSA-DQP builds upon parallel database technology [44,45] that supports both pipelined and partitioned parallelism. OGSA-DQP harnesses computational resources

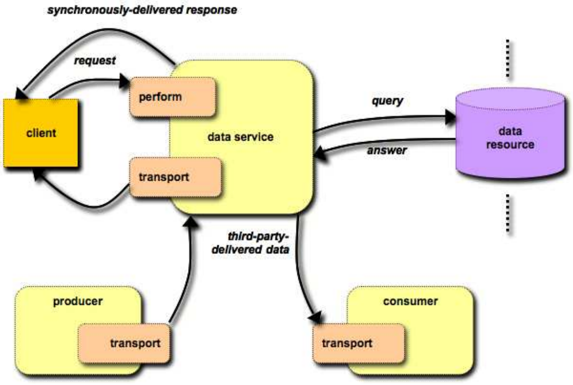


Figure 2. A Data Service and its functionalities.

commensurate (as decided upon by the query optimizer) with the estimated magnitude of the computational effort involved. In this sense, OGSA-DQP follows the Grid vision of utility computing, i.e., one in which heterogeneous, distributed, autonomous resources can be discovered, harnessed and combined to meet requirements as and when they arise. In this respect, OGSA-DQP is therefore most useful when queries involve computationally expensive operations (e.g., when the query contain many, or few particularly expensive, joins or else when it requires a call to a computationally-intensive external function) that can benefit from being executed in parallel.

1.4. Virtualizing data resources

We note that, in OGSA-DQP, data resources are virtualized using the OGSA-DAI (Open Grid Service Architecture - Database Access and Integration) middleware [2]. However, in the remainder of this section, rather than use the specific terms and describe the specific design choices taken in OGSA-DAI, we cast our descriptions at a slightly higher abstraction level with a view towards avoiding the undesirable consequence of coupling the research issues and challenges arising much too tightly to a particular software implementation that is, in any case, still being actively developed and continuously improved and enriched. We therefore encourage the reader to use [2] as a complement to this section.

Figure 2 depicts the most important functionalities exposed by the process of virtualizing a data resource; such resources appear in the leaves of OGSA-DQP plans. The most important service invocation points are perform and transport. A client is any software entity that submits a request to the data service. A request is typically a sequence of activities (including, of course, query evaluation) conveyed in document form that, because of its destination point, is often referred to as a perform document. The OGSA-DAI implementation of a data service can link together (through the chain of inputs and outputs) the sequence of activities conveyed in a perform document. The response associated with a request can be synchronously-delivered to the client upon return, or else it can rely on the transport facilities for asynchronous retrieval of results. Such transport facilities can also be used to implement third-party delivery of data. Thus, a producer service can use its own transport mechanism to feed data into a data service and, correspondingly, a data service can feed data into a consumer service using the latter's transport facilities.

Naturally, the most fundamental role played by a data service is that of virtualizing a data resource by providing insulation against a range of heterogeneities. Thus, an activity may submit a query to be evaluated in order to obtain an answer set for it without much knowledge of the physical realization of the underlying data resource. The OGSA-DAI implementation of this capability ranges over structured, semi-structured and unstructured data resources (in particular, relational DBMSs, XML data repositories and classical files). In this way, the virtualization process offers a degree of insulation not only against the complexities of negotiating access to specific software and hardware infrastructure, since it also enables requests to be phrased one abstraction step above that, which would lead considerations as to the nature of the data resource it virtualizes to be required.

1.5. Virtualizing distributed query execution

OGSA-DQP introduces two new kinds of service, viz., coordinator services and evaluator services. A coordinator service uses extensibility points provided by an OGSA-DAI data service to support compilation, optimization, partitioning and scheduling of queries for execution. An evaluator service corresponds to a query evaluation engine in a classical, centralized DBMS, i.e., it takes (possibly a fragment of) a query execution plan and evaluates it. Within an evaluator, classical algebraic operators are evaluated using pipelined parallelism insofar as every algebraic operator instantiates an iterator pattern [23]. As a result, an algebraic operator tree is a tree in which every non-leaf node is an iterator running its own thread of control. Partitioned parallelism is captured by cloning query plan fragments that execute in parallel over data partitions of the original input(s). Both communication and data partitioning are delegated to an exchange operator [22].

As hinted above, a distributed query execution plan maps to an orchestration of evaluator services rooted on a coordinator service and grounded on data services and analysis services. There are two phases to DQP using the OGSA-DQP approach. In the one-off first phase, the client sets up a distributed query service with a document that describes the schemas to be imported in order to form the federation (i.e., the component data and analysis services) and configures the capabilities of the latter (principally by assigning to it a pool of evaluator services over which the distributed query execution plan fragments can be scheduled). The first phase follows a factory pattern and hence it has as its outcome the return of a handle that can be used to invoke the instantiated distributed query service. In the second phase, once the distributed query service is set up, it can be used to evaluate queries over the virtual federated database for however long the distributed query service is meant to be active for.

Figure 3 depicts the most important interactions that characterize the second phase alluded to above, i.e., the figure illustrates the compile time and the run time behaviour of OGSA-DQP. When a client wants to evaluate a query, it requests the evaluation of the latter by means of a perform document sent to the distributed query service for which the handle is held. The compilation, optimization and scheduling of a query is described in more detail later in this section. Its outcome is the harnessing of as many evaluator services as required by the fragmentation decisions made. In Figure 3, as an example, four evaluator services were harnessed and there are three leaves, viz., two data

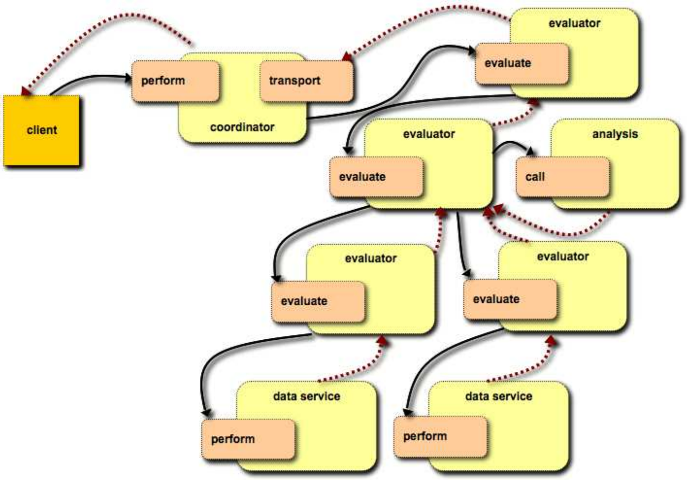


Figure 3. DQP Interactions: the compile time and the run time behaviour of OGSA-DQP.

services and one analysis service. Recall that query execution in OGSA-DQP amounts to the execution of an orchestration. Thus, to the operator tree comprising iterator nodes, there corresponds an orchestration such as exemplified in Figure 3. Execution proper begins by invocation of the evaluate capability at the root of the orchestration. Because of the iterator semantics it implements, this invocation cascades down to the children services until the orchestration reaches the leaves. Note that this process is denoted by the top-to-bottom solid arrows from the root to the leaves. As in classical query processing based on iterators, to this downward cascade of invocations there corresponds an upward cascade of data flows that culminate, at the root, with items being added to the result set. This process is denoted by the bottom-to-top dashed arrows from the leaves to the root. Note, furthermore, that the root delivers results to the coordinator using its transport capability.

The compilation, optimization and scheduling of a query is structured into a two-phase approach as follows. The query is first parsed and type-checked to yield a logical query plan (in the form of an algebraic operator tree). This is rewritten (on the basis of selectivity-driven estimates of intermediate result size) into a canonical form to obtain a heuristically-efficient join order. A cost model then drives the assignment of concrete join algorithms to the logical joins that occur in the canonical form. This concludes the first phase: its outcome is a single-node query execution plan, i.e., one in which there is no plan fragmentation and hence no need for inter-fragment communication either. The second phase then takes over with the goal of partitioning the single-node query execution plan into fragments. In this phase, the optimizer considers whether parallelizing joins or calls to analytical services could lead to significant speed-ups. Finally, scheduling decisions are taken in the light of the characteristics (such as available memory) of the computational resources that were imported as metadata when the federation was configured at set-up time.

Given two data services that use OGSA-DAI to virtualize a protein sequence database and an ontology (e.g., GO) on biological terms, and one analysis service virtualizing a BLAST sequence comparison server, one might make use of them by means of a query such as

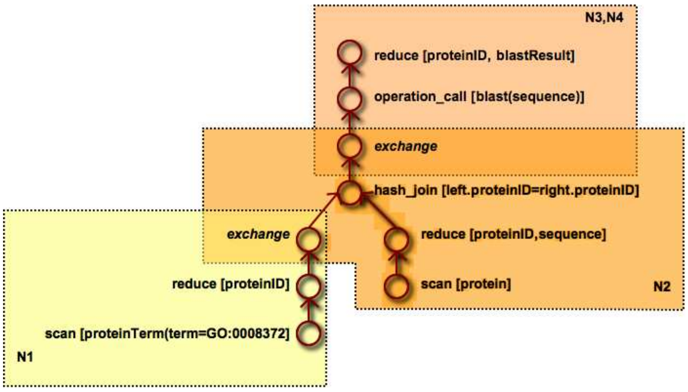

N3,N4
reduce [proteinID, blastResult]
operation_call [blast(sequence)]
exchange
hash_join [left.proteinID=right.proteinID]
exchange
reduce [proteinID,sequence]
reduce [proteinID]
scan [protein]
N2
scan [proteinTerm(term=GO:0008372]
N1

Figure 4. An Example Distributed Query Execution Plan

```
select p.proteinID, blast(p.sequence) as blastResult
from Protein p, Ontology o
where p.proteinID = o.proteinID
and o.proteinTerm="GO:0008372"
```

Figure 4 depicts one possible outcome of the compilation, optimization and scheduling of the above query[2] over for computational nodes named N1, N2, N3, and N4. Note that Figure 4 reflects decision to run the call to BLAST with degree of parallelism equal to two, while the hash-join has not been parallelized.

1.6. Open and Research issues

The emphasis of OGSA-DQP is on enabling querying over dynamically created federations of autonomous databases and supporting parallel execution of such, potentially intensive, distributed queries building upon a service-oriented architecture. In essence, the OGSA-DQP system makes a trade-off between flexibility and performance. Service-oriented solutions are very appealing in Grid environments where on the fly creation of *virtual organizations* takes place; nevertheless they are not always tailored to high performance applications because the message-based access method to services incurs a high communication cost. Significant effort has been put in improving performance. This effort is in two complementary directions. The first direction deals with the increased data transfer cost, and, in the second direction, novel approaches to efficiently employing parallelism, both at compile time and at runtime, are sought. The derived solutions are described in the sequel of the chapter.

The open research issues are actually broader and apply to all parallel query processors for the Grid (e.g. [30]). Grid is an inherently volatile and unpredictable environment. As a result, the information required to take optimal decisions, either with respect to query plan optimization (e.g., selectivities of operators over remote, autonomous databases) or with respect to resource allocation (e.g., current and average load of remote computations resources) may be inaccurate, or even, completely unavailable at compile

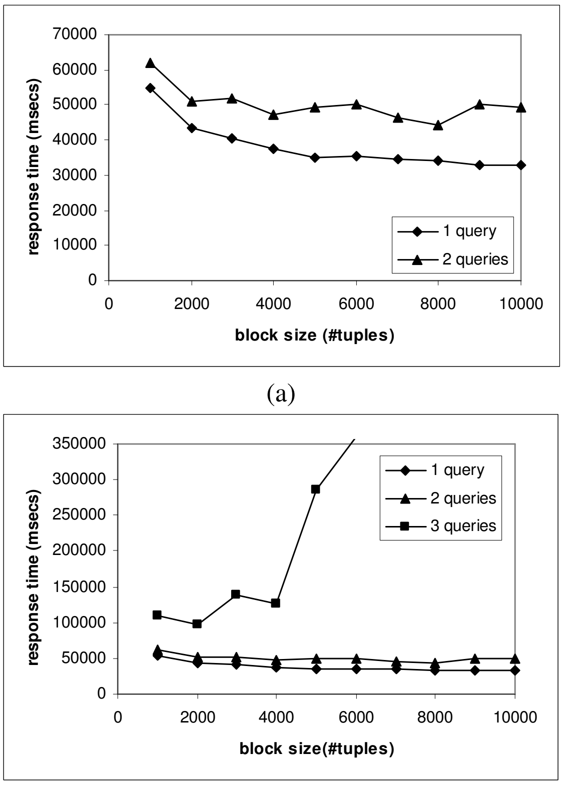
[2] In the OGSA-DQP physical algebra, *project* is referred to as reduce, an external call is represented by operation_call and the scan operator generalizes the sequential and indexed versions of that operation.

time and only known at runtime. In this case, the compile time decisions should be continuously refined at runtime, in the light of the new information when this becomes available. In most cases, adaptive query processing (AQP) techniques are limited to a single form of adaptation [9] and refer to centralized processing. Thus, it is important both to extend AQP solutions for centralized processing so that they are applied to a Grid setting and to combine them. Combination of different AQP techniques is a rather overlooked subject. An adaptive extension of OGSA-DQP presented in [18] deals with combined forms of adaptations (i.e., performing load balancing and adjusting the degree of partitioned parallelism), but does not support runtime changes to the query plan. DQP systems for Grids should also be coupled with Grid monitoring systems [54]. Finally, in OGSA-DQP, certain simplifying assumptions are made: all data belong to reliable sources, sharing of query state is not permitted and the query processing objective is to evaluate a query as quickly as possible. Obviously, a more generic solution should include solutions for handling unreliable peers, performing more efficient multi-query optimization and applying multiple optimization criteria, such as monetary cost (e.g., as in Mariposa [47]) and resource utilization along with query response time, possibly defined through a Service Level Agreement (SLA) [41].

2. On mitigating the impact of increased data transfer cost

Although web/grid services provide one of the most practical solutions for accessing remote data sources and programs spanning multiple administration domains, the fact that they are slow (due to high communication costs and parsing overheads) makes their use in high performance applications problematic. Minimizing the data transmission cost for queries over web services is of high significance for modern parallel and distributed database applications, because, when data transmission cost dominates, parallel solutions may not scale well and become inferior to non-parallel ones. A mechanism that can mitigate the impact of high data transfer cost is to pull data from a service-wrapped database (e.g., [2]) or submit calls to a service to perform data processing (e.g., [46]) in a block-based manner. In this mode, the data transmission cost per se can decrease, whereas applications can also benefit from pipelined parallel processing, i.e., the actual transmission cost both decreases and is easier to be hidden as it overlaps with computation cost.

Consider a client that submits a data retrieval task to an OGSA-DAI wrapped MySQL DBMS. The client resides on a randomly chosen PlanetLab node [6] in Switzerland, and the retrieval task consists of an inexpensive (in order not to incur significant CPU load) scan-project query over the entire *Customer* relation of the TPC-H benchmark, when this is deployed with scale set to 1, i.e., in total, 150K tuples are retrieved. The server is in the UK. This query is executed in a pull mode, i.e., the client continuously requests for new data chunks that can have different size each time, until the complete result set is retrieved. The main challenge in optimizing the chunk-based data transmission stems from the fact that the graph describing the aggregate transmission cost of a given dataset in time units with regards to the block size, has a concave shape with various local minima and high noise. Figure 5(a) shows the performance degradation and the increased concavity of the response time graph with regards to the block size, when the client submits two queries at the same time. The quadratic effect is even more obvious when there are three concurrent queries and the server received more load in



(a)

(b)

Figure 5. The response times when (a) 2 queries and (b) 3 queries are being responded concurrently (figure from [21]).

terms of memory utilization between the second and the third query (see Figure 5(b)). In this case, not only the shift of the optimum size is larger, but the effect of a suboptimal decision can be detrimental. For example, under these conditions, if the optimum size for two concurrent queries is chosen, and one more query is submitted, then, for the same block size, the response time would be an order of magnitude higher than the optimum.

In [20], adequate evidence is provided that, in practice, the optimal point changes for each query submitted due to the differences in the tuple size and the network and server conditions at each time point. Moreover, the optimal points changes during query evaluation, as well. In this section[3], we present adaptive solutions to the problem of finding the volatile optimum block size, with a view to providing robust runtime solutions for minimizing query tasks that involve the transmission of large datasets to and from services. These solutions are minimally intrusive; they operate at the client site, and thus require no additional monitoring of and extensions to the servers.

The approach followed is a control theoretical one. The problem under investigation falls into the broader vision of developing autonomic, self-managing solutions for data management [33]. In principle, autonomic computing can benefit a lot from con-

[3] This section is largely based on the work in [20,21].

Algorithm 1 Pseudo-code at the client side

```
blockSize = initialBlockSize
**while** !end-of-results **do**
    $t_1 = timestamp()$
    $WebService.requestNewBlock(blockSize)$
    $t_2 = timestamp()$
    $blockSize = Controller.computeNewSize(t_2 - t_1)$
**end while**
```

trol theory techniques, which are well-established in engineering fields and are typically accompanied by theoretical investigations of properties such as stability, accuracy, and settling time. Applying techniques inspired by control theory to the complex problems of distributed computing is not something new; actually there is a recent booming in development control theoretical approaches to solve problems in computing systems, software engineering and software services. This is due to the trend of going beyond ad hoc and heuristic techniques towards an autonomic computing paradigm [11]. Exploitation of the rich arsenal of techniques, methods, ideas and foundations of control theory, developed for many decades since the second world war, has already led to improved designs in many areas and problems, such online adjustments of web, application and database servers.

In control systems, the main part is the controller, which receives system measurements, and provides a system configuration that impacts on system performance. There are several orthogonal dimensions across which controllers can be characterized and compared. A controller can be either single-input or multiple-input, and similarly, single-output or multiple-output. The controller inputs are adjustable configurations of key system parameters, such as block size or proportion of tuples or memory allocated to a machine, whereas the outputs are measurable properties of the system, such as response time, CPU utilization, and so on. When the measured outputs impact on the controller input, then the controller is termed as feedback or closed-loop, otherwise it is called feed-forward or open-loop. The former does not require the development of accurate complex models and can tolerate relatively high model inaccuracies, thus it is more practical for use in volatile environments. The controllers can be either continuous-time or discrete-time; mostly all controllers of computing systems belong to the latter category due to the nature of computing systems. Finally, controllers can be adaptive themselves in the case where runtime measurements can lead to changes in their own design. In our case, the controller is a single-input single-output feedback one, which receives as input the response time of each block size. Based on this value, it decides the size of the next block to be pulled from the service that wraps the database. More specifically, the solutions developed are inspired by *extremum control* [3], which can yield results and track a varying optimum operating point even in the absence of a detailed analytic model.

2.1. Solution description

To provide the *extremum control* solution to this particular runtime optimization problem, we adopt a control-theoretical approach, according to which a lightweight controller is encapsulated in the client (see Algorithm 1). *Extremum control* [3] can yield results and track a varying optimum operating point even in the absence of a detailed analytic model; it is based upon numerical optimization but goes beyond that since it can be blended with well known control approaches, including variable setpoint (optimum tracking) controllers, feedforward controllers, perturbation analysis, self tuning and adaptive techniques, so that noise, model uncertainties and time variations can be dealt with. Filtering and averaging are also typically included in the aforementioned techniques.

Let y be the performance metric, such as response time or, equivalently, the per tuple cost in time units, and x denote the size of the data block. Then, there is a typically unknown function f, such as $y = f(x)$. The role of an extremum controller is to manipulate the input x to the performance function $f(x)$, as a function of this output. In switching extremum control, the value of x at the kth step, x_k is given by the following formula:

$$x_k = x_{k-1} - g \cdot sign(\Delta y_{k-1} \cdot \Delta x_{k-1}) \tag{1}$$

where $\Delta u = u_k - u_{k-1}$. The function $sign()$ returns 1 if its argument is positive and -1 otherwise. g corresponds to the gain and can be either constant or adaptive. The formula above can detect the side of the optimum point where the current block size resides on. The rationale is that the next block size must be greater than the previous one, if, in the last step, an increase has led to performance improvement, or a decrease has led to performance degradation. Otherwise, the block size must become smaller.

Several heuristics are applied to enhance the performance switching extremum control. To mitigate the impact of the noise in the graphs, the measured output and the control input are firstly averaged over a sequence of n measurements. However, there is a trade-off between noise removal and speed of response. In addition, maximum and minimum limits can be imposed to avoid overshooting with detrimental effects, as there is no guarantee that the controller will not reach a very high or very low value before converging. Finally, to facilitate the controller to be capable of continuously probing the block size space, since the optimum point may move during query execution, a dither signal may be added.

Complementarily, another direction can be explored, after observing that, despite the volatility, local peaks, jitter etc., the performance graphs can be represented by smooth quadratic (or sometimes monotonically decreasing) concave curves. In this direction, runtime system identification techniques are developed to fit the online data to smooth profiles. The latter can subsequently be processed analytically. For example, the curves describing the response time as a function of the block size can be approximated as quadratic or parabolic functions. Then, the client submits several test calls to the service with different block sizes (from rather small ones to rather large ones), so that a small set of samples is created. These samples can be used to estimate the model parameters by applying least-squares method.

2.2. Discussion

The extremum control-based techniques can be distinguished between techniques employing constant gain g in Eq. (1) and those employing adaptive gain, proportional to the performance changes due to the last decisions. The trade-offs between these two types can be summarized as follows (the interested reader can refer to [20] for more details). Adaptive gain policies seem to be the most suitable choice when the near optimal region

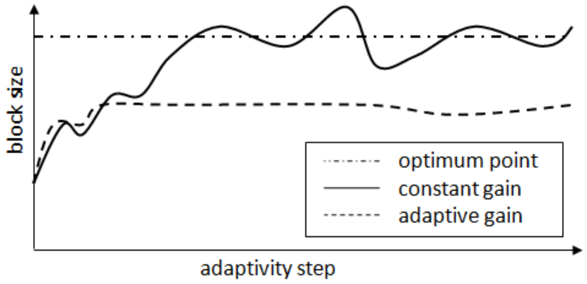


Figure 6. Comparison of the behavior of constant and adaptive gain switching extremum controllers.

can be approximated. However, in this case the performance benefits may not exceed 10% decrease in response times. Larger improvements, over 100% decrease in performance degradation, can be provided when this region is not a priori known. In this case, adaptive gain policies have nice transient (i.e., they converge quickly) and stability (i.e., they do not oscillate after convergence) properties but their accuracy and capability of convergence are quite sensitive to noise and non-smooth profile shapes (i.e., they do not converge always). On the other hand, constant gain policies can perform well even without proper tuning, but their transient behavior and steady state stability can deviate from the optimum point. In summary, both approaches improve on the static case, where decisions on the block size are taken at compile time and are not subject to runtime modifications, but none of these two types of switching extremum control is robust with respect to different settings. Figure 6 depicts the typical behavior of the two flavors when the initial block size is not close to the optimal. Other adaptive approaches that can be followed include Newton-based hill-climbing techniques. However, such techniques are not suitable for systems exhibiting noisy, non-monotonical behavior, and are outperformed by switching extremum control [20].

The severe limitation regarding the controller robustness is addressed in [21], which proposes a novel hybrid controller that aims at combining the strengths of both constant and adaptive gain controllers, with a view to improving robustness and average performance. The hybrid controller is still described by the model in Eq. (1), but the gain is now defined as follows:

$$g = \begin{cases} b_1, \text{in transient phase} \\ b_2 \| \frac{\Delta y_{k-1}}{\bar{y}_{k-1}} \Delta x_{k-1} \|, \text{in steady-state phase} \end{cases} \tag{2}$$

In the hybrid model, while trying to detect the optimal region, the gain is a constant (positive) tuning parameter, so that this region is detected both accurately and quickly. After converging, the gain is switched to adaptive and proportional to the product of the performance change and the change in the block size. In this way, oscillations are avoided. The drawback is that a mechanism for deciding whether steady-state phase has been reached must be developed. To this end, building upon the observation that a constant gain switching extremum controller oscillates around the stability point in a saw-tooth manner, the following criterion can be applied, which operates over a horizon of length n (s is a small integer, and, k the step number):

static 1K	static 10K	static 20K	const. gain	adapt. gain	hybrid	best model
53.3%	81.5 %	226.8 %	21.3 %	37.5 %	13.5 %	0.7 %

Table 1. Performance degradation for different approaches to block size selection (from [21]).

$$\| \sum_{i=k-n}^{k-1} sign(\Delta y_k \cdot \Delta x_k) \| \leq s \tag{3}$$

The performance of the hybrid controller is superior to simple constant and adaptive gain controllers. However, the performance can be further improved if combined with model-based solutions as reported in [21], where evidence is provided that the performance can be similar to the case that the optimal block size is *a-priori* known. More specifically, in Table 1, the adaptive solutions are compared against three static cases, for several representative cases, as explained in [21]. The three static cases correspond to blocks of fixed size: 1K, 10K and 20K tuples, respectively. For the model-based solutions, both quadratic and parabolic models were examined. Interestingly, if a self-configuring mechanism was in place to decide at runtime which is the best performing model in each case (i.e., either the quadratic or the parabolic one), then the performance would be very close to the optimal. Note that the construction of the models is rather inexpensive compared to SOAP-based data transmission.

Nevertheless, there are some open issues remaining. In particularly long-running queries, the near-optimal region may change during execution significantly . This phenomenon impacts on both the hybrid controller and the model-based solutions. The hybrid controllers may need to extend and enhance their switching state criterion to detect cases when the controller should re-enter a transient phase. Also, model-based solutions should periodically refine their estimates, possibly using methodologies such as recursive least squares. Finally, complementary efforts to minimize the data transfer cost are described in [42] and [28]. The former suggests improvements to the basic communication mechanism for web services, whereas the latter investigates solutions based upon runtime selection of the transfer protocol. A promising avenue for future work is to combine all these different approaches into a single unified solution.

3. Scheduling parallel queries over non-dedicated distributed resources

3.1. Background and motivation

OGSA-DQP provides middleware parallel DQP functionality, rather than database solutions to specific Grid problems, as in, e.g., [37]. As such, it can be used by different applications, each with its own computational requirements. In order to be capable of efficiently supporting all kinds of applications (i.e., both the less and the more computationally demanding), OGSA-DQP can support arbitrary degrees of parallelism. Parallelism is inserted in the query plans through the exchange operator [22]. By default, the evaluator services that are responsible for retrieving data from their source are responsible for the evaluation of the rest of the query plan as well; however, any node in this plan can be cloned as many times as there are machines available [17]. Scheduling parallel queries in Grids involves issues that are not applicable to traditional DQP and parallel query processing[4].

Note that the problems of defining the execution order of subplans and exploiting pipelined parallelism are addressed by adopting well-established execution models, such as iterators [23], and thus, need not be part of query schedulers. In other words, in query processing in which the tasks (either at the level of subplans or at the level of individual operators) are interdependent, both inter-operator and intra-operator parallelism are applied as explained previously; however the scheduling problem deals mostly with intra-operator parallelism. More specifically, the resource scheduling problem in databases for the Grid is reduced to a resource allocation problem of (i) choosing resources and (ii) matching subplans with resources. Existing scheduling algorithms and techniques, either from the database or the Grid or the parallel research communities, seem 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. Generic directed acyclic graph (DAG) schedulers (e.g., [48]), and their Grid variants (e.g., [49]) tend to allocate a graph vertex to a single machine instead of a set of machines, which leads to no partitioned parallelism. More comprehensive proposals (e.g., GrADS [7]) still rely on application-dependent "mappers" to map data and tasks to resources, and thus come short of constituting complete scheduling algorithms. Other proposals for mixed-parallelism scheduling (e.g., [39]) and parallel database scheduling (e.g. [12,10]), are restricted to homogeneous settings. For all these reasons, simple solutions, such as *minimum completion time*, *opportunistic load balancing*, *min-min*, *max-min*, are not useful.

Trivial solutions, such as choosing the maximum degree of parallelism, not only harm the efficiency of resource utilisation, but can also degrade performance, similarly to what holds in homogeneous settings [51]. In general, 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, leaving aside the problem of devising optimal workload distribution among the selected machines. 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 difficulty has to do with the efficiency of parallelisation, which 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 [32], effective and efficient heuristics need to be employed, like the one proposed in the sequel.

3.2. A novel scheduling algorithm

[4] This section is based on material from [16].

Algorithm 2 High level description of the scheduling algorithm.

```
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 complexity of the problem of resource selection and scheduling on Grids justifies resorting to heuristics, as an exhaustive search for all the possible combinations of machines, workload distributions and query subplans is an obviously inefficient solution. An acceptable solution will be one that can scale well with the number of machines that are available.

The algorithm proposed here receives a query plan which can be partitioned into subplans that can be evaluated on different machines. For the shape of the query plan we assume the existence of a query optimiser which first constructs a single-node plan, and then transforms the single-node plan into a multi-node one, in order to reduce the search space. This method has been widely adopted in parallel and distributed query processors [29], as it performs well in a wide range of cases [8]. The scheduler that will be presented deals with the second stage, where the single node plan is parallelised. As such it extends and does not replace optimisers that produce non-parallelised query plans. The algorithm operates at compile time, i.e., it is static.

The algorithm follows a hill climbing approach. Initially, each of the operators of the query plan is scheduled on one machine, i.e., the initial condition a query plan with minimum partitioned parallelism; such a query plan is unlikely to perform well for intensive computations. After this initial resource allocation, it enters a loop as shown in Algorithm 2. In that loop, the algorithm estimates the cost of the query plan and of each physical operator of the query plan individually. Then, it takes the most costly operator that can be parallelised, and defines which criteria should be used for selecting machines for this operator. For example, some operators place more importance on memory requirements than others, whereas other operators may be more computationally and communication intensive. Following this step, a greedy procedure is performed in which the algorithm increases the operator's degree of parallelism by one degree if that increase improves the performance of the query plan above a certain threshold. 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, and the system proceeds immediately to the execution phase. Full details are provided in [16]. 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.

To estimate the cost of the query plan and individual operators, the algorithm requires a decoupled cost model which (i) assigns a cost to a parallel query plan, and (ii) assigns a cost to every physical operator of that query plan. Any such cost model is suitable, as the scheduler is not based on any particular one. In other words, although the algorithm depends on the existence of a cost model, it is cost model-generic. By decoupling the cost model and the scheduler algorithm, enhancements in both these parts can be developed and deployed independently. The cost model is also responsible for defining the cost metric, which directly corresponds to the optimization criterion. As such, the scheduler algorithm is independent of any specific optimization criteria.

This proposal effectively addresses the resource scheduling problem for Grid databases in its entirety, allowing for arbitrarily high degrees of partitioned parallelism across heterogeneous machines. The practicality of the approach lies in the fact that it is not time-consuming. The algorithm comprises two nested loops. 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. Its main novelty is that it does not restrict the degree of intra-operator to any extent and does take into account the fact that the resources available are heterogeneous, along with empirical evidence that the algorithm leads to performance improvement and robustness.

Empirical evidence is based on experiments against other typical choices. These choices include scheduling that use all machines available, or they do not support partitioned parallelism, or they use only machines that store data, or they use only the top-k machines in terms of CPU power, or they parallelise only the most expensive operator [16]. More specifically, the evaluation presented in [16] showed that the approach yields query response time improvements of several factors 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.

3.3. Discussion

The scheduler algorithm presented previously is the first step towards efficient and effective scheduling solutions for DAGs that are executed on heterogeneous nodes with arbitrary degrees of partitioned parallelism. It follows a hill climbing approach, where the degree of parallelism of each operator is decided separately in a greedy manner.

Nevertheless, it is characterized by several weak points that can be regarded as directions for future work. First, no formal bounds on its worst case performance have been defined; defining such bounds is a difficult but necessary issue in order to supplement the complexity analysis of the algorithm. Second, it is unclear what is the behavior of the algorithm when it is called at runtime and whether it can lead to instability if applied dynamically during query evaluation. In addition, although the algorithm scales linearly with the number of available machines, in a Grid setting, this number may be too high. So, it may be desirable to develop algorithms that consider only a subset of such machines in order to reduce complexity. Efficient selection of such subsets is an open issue.

Finally, there are several issues related to the cost model employed by the algorithm. A suitable cost model for Grid settings should not only model efficiently all forms of parallelism, but also, it should be robust to uncertainties of machine characteristics. More importantly, it should be able to support multi-objective scheduling in a clear manner, which also requires extensions to the basic scheduling algorithm.

4. Balancing parallel queries over non-dedicated distributed resources

In the previous section, a scheduling algorithm was presented that is capable of efficiently allocating resources for parallel query processing on Grids. Its key aspect is that it allows queries to benefit from intra-operator parallelism, which may be of a different degree for different subplans. However, when the optimization criterion is query response time and in order to fully exploit the potential of partitioned parallelism, work needs to be assigned to machines in a way that reflects their capabilities, which is challenging in an environment with heterogeneous and potentially autonomous, non-dedicated resources. Key characteristics of a typical such environment include unpredictable fluctuations in the load of available machines. A consequence of this fact is that it is not efficient to divide a task into several partitions and to stick with this division until completion. In addition, the information about the machine characteristics that describe performance capacity and loads is typically incomplete and/or inaccurate at compile time; thus a load balancer with responsibility for efficient work assignment should rely mostly on runtime feedback. Finally, due to the fact that the instances of subgraphs are stateful, the cost of workload re-assignments, which typically depends on the size of the state, is not negligible in general.

One of the most notable examples trying to address the aforementioned characteristics is the Flux approach [43], which introduces a new operator that monitors the execution speed and the idle time of each participating machine at runtime, and adjusts the workload allocation accordingly, with a view to equalizing machine utilization. Additional heuristics are applied to smooth the workload allocation changes. Flux tries to guarantee that the time spent enforcing adaptivity decisions (i.e., moving state from one machine to another as a result of a workload reallocation) does not exceed the time of query processing; this is done by keeping the same workload allocation for a period that is at least equal to the time spent carrying out the adaptation that brought it about. Note that this heuristic may not always prevent performance degradation. Indeed, Flux may adapt in a non-beneficial manner in response to transient and periodic imbalances, as it may keep shifting the state partitions.

Several attempts have been made to improve the behavior in these situations. In [38], some extensions to the Flux approach are described. More specifically, a change to the Flux algorithm is proposed, to carry out replication during the probe phase in operators involving hash tables. In other words, the adaptivity decisions are taken in the same way but the operator state is not moved as in Flux, rather it is replicated at the expense of higher memory usage. This may reduce the number of future state movements, and, consequently, it performs significantly better when many adaptations are needed during execution, assuming that there are no memory limitations. As such, it may be quite successful in some cases, but is suboptimal where the build phase is long compared with the probe phase, or where memory is not abundant, or where the operators are window-based. In general, its application space is rather narrow. Another proposal is the use of a dynamic hash table approach in which all hash table inserts and probes take place twice. However, this seems not to be a winner, since it is characterized by a significant response time overhead where no load balancing is required, and by creating considerable amounts of extra work reduces throughput in a loaded environment.

Typically, the load balancing problem in a single query environment is transformed to the problem of making the execution times of parallel subtasks equal, as their maxi-

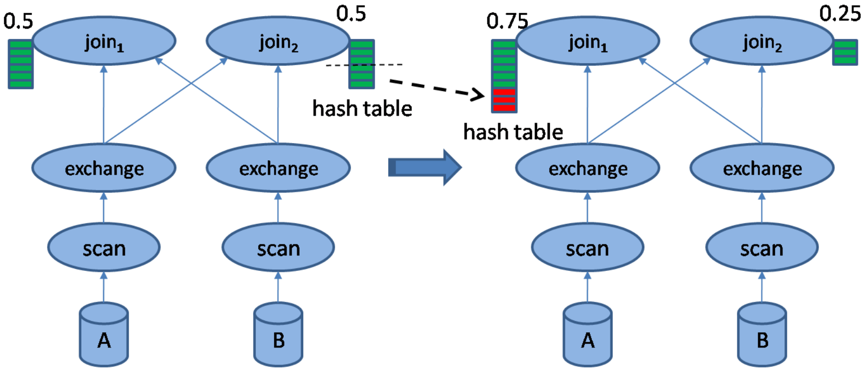


Figure 7. An example of balancing stateful operators.

mum defines the overall execution time.The spirit of Flux is the same, although the adaptivity steps are not based on a corresponding balancing function, but on a heuristic, as mentioned previously. Such approaches essentially adopt a definition of balanced execution, which does not take into account the inherent overhead for enforcing the balancing decisions. This limitation, which is particularly felt in unpredictably volatile imbalances, needs to be addressed; otherwise, any load balancing approach will not perform well under randomly and rapidly changing load imbalances. In the sequel, the load balancing problem is formalized in a way that the enforcing overhead is directly taken into consideration.

4.1. Problem Description

Consider two relations, *A* and *B*, which are joined remotely using a hash join; the hash table is built on *A*. Let us assume that the hash join operator is parallelised over two physical nodes, and that these two nodes are capable of processing tuples at the same speed. Then, in a balanced execution, the two nodes should receive and process the same amount of workload. However, if, during execution, the first machine becomes three times as fast as the second machine, then the workload distribution should change to reflect that. The problem is that a workload distribution that is proportional to the nodes' execution speed can yield the lowest response times only if the operators are stateless. In stateful operators, like hash joins, which create internal state in the form of hash tables, any workload re-allocation triggers state movements, which incur some cost (see Fig. 7). Consequently, a more efficient load balancer should take into account this cost when deciding on workload re-allocations with a view to reducing the query execution time.

The load balancing problem can be formalized as follows. Let P be the degree of intra-operator parallelism of an operator o, and $m_1, m_2, \cdots, m_P$ the P nodes participating in its execution. The workload proportion that each of these nodes receives at the kth adaptivity step is $p_1(k), p_2(k), \cdots, p_P(k)$, with the constraints $\sum_{i=1}^{P} p_i(k) = 1, \forall k$ and $p_i(k) \geq 0, \forall k$. $p_i(k), i = 1 \ldots P$ is defined through a hash function $h_k()$. Each node possesses a certain amount of state $s_i(k), i = 1 \ldots P$, which is needed to evaluate $p_i(k)$. $s_i(k)$ depends on $p_i(k)$. $c_i(k)$ denotes the cost (overhead) to reach state $s_i(k)$ from state $s_i(k-1)$, as a result of a change in $p_i(k)$. The measured output is

$y_1(k), y_2(k), \ldots, y_P(k)$ and defines the expected value for the completion time of each of the participating nodes given the workload allocation of the kth adaptivity step. If the query is continuous (e.g., over data streams) the expected completion time refers to the time to complete the evaluation of a fixed aggregate workload. Without loss of generality, we can assume that $y(i)$ strictly monotonically increases with $p(i)$, i.e., assigning more workload to a node leads to an increase in its expected completion time. The role of the load balancer is to minimize the following

$$max(y_i(k+1) + c_i(k+1)), \;\; i = 1 \ldots P \tag{4}$$

and estimate h_{k+1} accordingly. To date, there is no practical methodology to solve Eq. (4) analytically. Empirical evaluation proves that approaches such as Flux, or the one described in [17] are efficient if the load imbalance is static. But when the imbalance changes frequently and unpredictably, these approaches are unsuitable.

4.2. A control theoretical approach

Essentially, the load balancing objective defined in Eq. (4) includes a trade-off between (a) reaching the optimal workload allocation, in which the expected completion times are equalized across all participating nodes, and (b) the cost for reaching such an allocation, which is mainly due to state movements. In addition, rapidly changing applications are characterized by non-negligible dynamics, which can be addressed by control theoretical approaches. There exist a control theoretical approach that is particularly tailored to trade-offs such as the one mentioned above. More specifically, to meet the objective in Eq. (4), we can employ a state space model, with a linear quadratic regulator (LQR) [27] designed on top of it. LQR controllers are in general capable of accurately finding the controller settings that minimize a cost function, which can capture both the deviations from the optimal state and the cost to reach such a state. In essence, they do not try to postpone adaptations due to the cost they are expected to incur but to modify the response actions so that any adaptations applied are beneficial. Such an approach is described in [19].

Nevertheless, the load balancing problem can be deemed as a constrained optimization problem. In the LQR approach, the constraints are incorporated in an indirect way. There exist some other methodologies in the literature which treat constraints directly, such as suboptimal *constrained LQR* and *model predictive control (MPC) or receding horizon control* [35]. Orthogonally to this issue, any scalable adaptivity solution operating in distributed environments, must be distributed itself; as such, investigation of distributed MPC [4], may lead to significant improvements.

Finally, the solutions mentioned thus far in this section refer to intra-operator parallelism, whereas more global load balancing solutions should consider both intra- and inter-operator parallelism in a unified manner. This is a significant open research issue. The work in [18] can be seen as a first step towards this direction, although it suffers from serious limitations (e.g., no runtime changes in the shape of the query plan and the operator execution order are supported).

5. Spatio-temporal query processing

The fundamental goal of spatio-temporal data management is the representation, storage and processing of queries involving the spatial (e.g., location) and/or the temporal (e.g., time instance) characteristics of objects, effectively and efficiently [31]. Many real-world applications are characterized by spatio-temporal behavior such as: traffic monitoring, fleet management, land ownership, monitoring of physical phenomena (e.g., hurricanes, tornados) to name a few [52]. The basic property of the underlying data is that the location and/or the shape of the objects may change with respect to time.

In spatio-temporal applications, queries may be expressed by posing constraints on the spatial, temporal or spatio-temporal data characteristics. As an example, consider the following *ad-hoc* queries that may be expressed in a fleet management application: "*determine all trucks that have been passed through area 66 the last seven days*" or "*determine all trucks with similar trajectory to that of truck X*". In the aforementioned queries, both spatial and temporal characteristics have been used. To answer such queries efficiently, appropriate query processing techniques are required which are usually enhanced by sophisticated spatio-temporal indexing schemes [31]. In many cases, these indexing schemes are enhanced or extended versions of efficient spatial access methods that have been successfully applied in spatial-only applications [14]. In other cases, the time attribute is considered as just another dimension, enabling the direct utilization of conventional spatial access methods. Although such schemes may provide the necessary performance improvements for query processing, there is an additional requirement to keep the data up to date subject to frequent changes in their spatial characteristics. In addition to ad-hoc queries, which may involve past, present or future (predicted) data characteristics, some applications may require the processing of *continuous* queries. A continuous query remains active for a time period and its answer needs to be updated in a continuous manner. For example, the query "*show me the five closest vehicles for the next two hours*" is characterized as continuous, because its result should be updated every time there is a change. Continuous query processing requires different mechanisms than ad-hoc processing in order to guarantee efficiency and to avoid unnecessary computational costs.

In addition to the performance boosting offered by indexing, spatio-temporal data management, as any other data-oriented application, has a lot to gain by introducing parallelism. Although this alternative has been successfully used in the past, there is relatively little work performed in parallelizing spatio-temporal databases. Some of these efforts are briefly described in the sequel. Parallel processing of spatio-temporal queries is one of the major research directions in spatio-temporal data management, as it has been pointed out in [40]. In [24] the authors study the provisioning of spatio-temporal services in a shared-nothing environment composed of clusters of workstations (COW). One of the machines is the *master site* whereas the rest are termed *storage sites*. Although the performance of spatio-temporal query processing is improved, the usage of a master server introduces a single point of failure and it may also introduce bottlenecks. In [55] distributed algorithms have been studied to answer similarity queries among trajectories of moving objects. The space is partitioned into cells, and cells are assigned to different processors (or sites). A similarity query initiated by a querying node triggers the processing of the cells that intersect the trajectory of the query. The experimental results have demonstrated that the proposed techniques have significant performance improvements over the centralized method. However, this approach does not consider load balancing or adaptive query processing issues which are ubiquitous in a Grid environment. Moreover, there is no provisioning for handling frequent updates due to changes in the location of moving objects.

In [5] a framework has been proposed for processing historical spatio-temporal queries in sensor networks. Three algorithms have been proposed and their performance has been studied by performing a series of simulation-based experiments. The proposed framework assumes that each sensor stores its own data locally, and therefore there are limited possibilities for load balancing, which is considered very important in non-centralized architectures. Moreover, in sensor network applications, low energy consumption is considered more important than query response time, towards increasing the life-time of sensors.

Regarding continuous query processing, the work in [53] studies a framework for distributed processing of continuous spatio-temporal range queries. The proposed model framework is based on PLACE [36]. The space is divided to regions, and each region is covered by a regional PLACE server. The study reports results for range queries only. An important extension to this work is the consideration of other important queries like k-nearest-neighbors and joins. Another interesting work in this area has been carried out in [15]. The novelty of this approach is that moving objects are capable of performing computations and therefore, the load posed to each server is reduced significantly. This idea can be applied when each moving object is equipped with devices like PDAs, which can perform simple spatio-temporal computations.

Parallel Grid architectures offer significant flexibility in query processing, allowing sites that are autonomous and heterogeneous to participate in query execution in a uniform and straightforward manner. However, issues like load balancing and adaptive query optimization should be tackled towards efficient response times. Spatio-temporal data are characterized by frequent updates, large volumes and costly operations (e.g., spatio-temporal joins). These features require efficient query optimization algorithms and load balancing techniques to guarantee acceptable performance. So far, these issues have been partially studied on specialized environments, like parallel processors, clusters of workstations, multi-disk systems and sensor networks. Although these architectures show significant similarities with parallel Grids, the challenges posed by the latter require a more thorough investigation of performance evaluation issues. The performance of spatio-temporal data management algorithms should be evaluated by using meaningful and reliable benchmarks. Although such benchmarks have been proposed for conventional spatio-temporal systems [50], there is a lack of benchmarks for spatio-temporal data managements in parallel Grid architectures.

6. Conclusions

This chapter describes a parallel query processing system for Grids, namely OGSA-DQP and emphasizes on several complementary research issues that have a strong impact on the performance of DQP systems running on Grids. These issues include reduction of the data transmission cost, advanced scheduling strategies and load balancing policies for distributed, heterogeneous and rapidly evolving settings. Several solutions to these issues are described. In addition, the chapter discusses spatio-temporal queries, which is an important class of advanced queries that are likely to be included in data- and computation-intensive scenarios running on Grids. Overall, parallel query processing on Grids seems to be a technology in evolution, and more advances are expected in the following years.

Apart from the aforementioned topics, there are several additional open issues. Perhaps one of the most interesting one is the blending of adaptive query processing techniques that respond to updated or modified information about data statistics with dynamic techniques that adapt to changes in the execution environment. Also, query optimization on the Grid should be multi-objective and possibly driven by SLAs. Constructing and enforcing SLAs is a far from trivial task and novel techniques are required to this end. Finally, since the Grid is an unpredictable volatile environment, autonomic solutions (especially self-configuring and self-optimizing ones) to all these problems are required. Control theory, which is tailored to rapidly changing situations, seems to be a promising tool to achieve that. Indeed, early experiences with control theoretical solutions in Grid data managing problems (e.g., [21]) are encouraging.

Acknowledgements

We would like top thank our coworkers who have contributed to the work described in this chapter, our understanding of parallel query processing in Grid settings and the implementation of the OGSA-DQP system, including Rizos Sakellariou, who also kindly offered suggestions to improve this chapter, Norman W. Paton, Paul Watson, Nedim Alpdemir, Desmond Fitzgerald, Steven Lynden, Arijit Mukherjee, and Jim Smith.

References

[1] M. N. Alpdemir, A. Mukherjee, N. W. Paton, P. Watson, A. A. A. Fernandes, A. Gounaris, and J. Smith. Service-based distributed querying on the grid. In *Proc. of 1st International Conference on Service Oriented Computing - ICSOC*, pages 467–482. Springer, 2003.

[2] M. Antonioletti, M. Atkinson, R. Baxter, A. Borley, N.P. Chue Hong, B. Collins, N. Hardman, A.C. Hulme, A. Knox, M. Jackson, A. Krause, S. Laws, J. Magowan, N.W. Paton, D. Pearson, T. Sugden, P. Watson, and M. Westhead. The design and implementation of Grid database services in OGSA-DAI. *Concurrency: Practice and Experience*, 17:357–376, 2005.

[3] K. J. Åström and B. Wittenmark. *Adaptive Control*. Addison-Wesley, Reading, MA, USA, 1995.

[4] E. Camponogara, D. Jia, B. Krogh, and S. Talukdar. Distributed model predictive control. *IEEE Control Systems Magazine*, 22(1):44–52, 2002.

[5] A. Coman, M.A. Nascimento, and J. Sander. A framework for spatio-temporal query processing over wireless sensor networks. In *Proceedings of the Data Management in Sensor Networks Workshop*, pages 104–110, 2004.

[6] D. E. Culler. PlanetLab: An open, community-driven infrastructure for experimental planetary-scale services. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[7] H. Dail, O. Silvert, 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.

[8] A. Deshpande and J. M. Hellerstein. Decoupled query optimization for federated database systems. In *Proceedings of the 18th International Conference on Data Engineering, 26 February - 1 March 2002, San Jose, CA*, pages 716–732. IEEE Computer Society, 2002.

[9] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[10] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - a high performance dataflow database machine. In *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings.*, pages 228–237. Morgan Kaufmann, 1986.

[11] Y. Diao, J. L. Hellerstein, S. S. Parekh, R. Griffith, G. E. Kaiser, and D. B. Phung. Self-managing systems: A control theory foundation. In *Proc of IEEE International Conference and Workshop on the Engineering of Computer Based Systems ECBS 2005*, pages 441–448, 2005.

[12] R. S. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational data base system. In *Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 31 - June 2, 1978*, pages 169–180. ACM, 1978.

[13] M. J. Franklin, A. Y. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Record*, 34(4):27–33, 2005.

[14] V. Gaede and O. Guenther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[15] B. Gedik and L. Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. *IEEE Transactions on Mobile Computing*, 5(10):1384–1402, 2006.

[16] A. Gounaris, R. Sakellariou, N. W. Paton, and A. A. A. Fernandes. A novel approach to resource scheduling for parallel query processing on computational grids. *Distributed and Parallel Databases*, 19(2-3):87–106, 2006.

[17] A. Gounaris, J. Smith, N. W. Paton, R. Sakellariou, A. A. A. Fernandes, and P. Watson. Adapting to changing resource performance in grid query processing. In *Data Management in Grids, First VLDB Workshop, DMG*, pages 30–44, 2005.

[18] A. Gounaris, J. Smith, N.W. Paton, R. Sakellariou, A.A.A. Fernandes, and P. Watson. Adaptive workload allocation in query processing in autonomous heterogeneous environments. *Distributed and Parallel Databases*, to appear, 2009.

[19] A. Gounaris, C. Yfoulis, and N.W. Paton. Efficient load balancing in partitioned queries under random perturbations. *Submitted for publication*.

[20] A. Gounaris, C. Yfoulis, R. Sakellariou, and M. D. Dikaiakos. A control theoretical approach to self-optimizing block transfer in web service grids. *ACM Transactions on Autonomous and Adaptive Systems, TAAS*, 3(2), 2008.

[21] A. Gounaris, C. Yfoulis, R. Sakellariou, and M. D. Dikaiakos. Robust runtime optimization of data transfer in queries over web services. In *24th International Conference on Data Engineering, ICDE*, pages 596–605, 2008.

[22] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *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.

[23] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[24] M. Hadjieleftheriou, V. Kriakov, Y. Tao, G. Kollios, A. Delis, and V.J. Tsotras. Spatio-temporal data services in a shared-nothing environment. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, 2004.

[25] A. Y. Halevy. Data integration: A status report. In *BTW 2003, Datenbanksysteme für Business, Technologie und Web, Tagungsband der 10. BTW-Konferenz, 26.-28. February 2003, Leipzig*, pages 24–29. GI, 2003.

[26] A. Y. Halevy, M. J. Franklin, and D. Maier. Principles of dataspace systems. In *Symposium on Principles of Database Systems PODS*, pages 1–9, 2006.

[27] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[28] T. Kosar and M. Livny. Stork: Making data placement a first class citizen in the grid. In *24th International Conference on Distributed Computing Systems ICDCS*, pages 342–349. IEEE Computer Society, 2004.

[29] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[30] N. Kotowski, A. A. B. Lima, E. Pacitti, P. Valduriez, and M. Mattoso. Parallel query processing for OLAP in grids. *Concurrency and Computation: Practice and Experience*, 20(17):2039–2048, 2008.

[31] M. Koubarakis, T. K. Sellis, A. U. Frank, S. Grumbach, R. Hartmut Güting, C. S. Jensen, N. A. Lorentzos, Y. Manolopoulos, E. Nardelli, B. Pernici, H. Schek, M. Scholl, B. Theodoulidis, and N. Tryfona,

editors. *Spatio-Temporal Databases: The CHOROCHRONOS Approach*, volume 2520 of *Lecture Notes in Computer Science*. Springer, 2003.

[32] Y. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[33] S. Lightstone, B. Schiefer, D. Zilio, and J. Kleewein. Autonomic computing for relational databases: the ten-year vision. In *Proc.of the IEEE Workshop Autonomic Computing Principles and Architectures (AUCOPA)*, pages 419–424, 2003.

[34] S. Lynden, A. Mukherjee, A. C. Hume, A. A. A. Fernandes, N. W. Paton, R. Sakellariou, and P. Watson. The design and implementation of ogsa-dqp: A service-based distributed query processor. *Future Generation Comp. Syst.*, 25(3):224–236, 2009.

[35] J. Maciejowski. *Predictive control with constraints*. Prentice Hall, 2001.

[36] M.F. Mokbel, X. Xiong, and W.G. Aref. Continuous query processing of spatio-temporal data streams in place. *Geoinformatica*, 9(4):343–365, 2005.

[37] M. A. Nieto-Santisteban, J. Gray, A. S. Szalay, J. Annis, A. R. Thakar, and W. O'Mullane. When database systems meet the grid. In *Conference on Innovative Data Systems Research CIDR*, pages 154–161, 2005.

[38] N. W. Paton, J. Buenabad-Chavez, M. Chen, V. Raman, G. Swart, I. Narang, D. M. Yellin, and A. A. A. Fernandes. Autonomic query parallelization using non-dedicated computers: an evaluation of adaptivity options. *VLDB J.*, 18(1):119–140, 2009.

[39] A. Radulescu and A. J. C. van Gemund. A low-cost approach towards mixed task and data parallel scheduling. In *Proceedings of the 2001 International Conference on Parallel Processing, ICPP 2002, 3-7 September 2001, Valencia, Spain*, pages 69–76. IEEE Computer Society, 2001.

[40] J.F. Roddick, M.J. Egenhofer, E. Hoel, D. Papadias, and B. Salzberg. Spatial, temporal and spatio-temporal databases-hot issues and directions for phd research. *SIGMOD Record*, 33(2):126–131, 2004.

[41] R. Sakellariou and V. Yarmolenko. Job scheduling on the grid: Towards sla-based scheduling. In Lucio Grandinetti, editor, *Volume 16 in the Advances in Parallel Computing series*, pages 207–222. IOS Press, 2008.

[42] B. Seshasayee, K. Schwan, and P. Widener. SOAP-binQ: High-performance SOAP with continuous quality management. In *24th International Conference on Distributed Computing Systems ICDCS*, pages 158–165, 2004.

[43] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 25–36. IEEE Computer Society, 2003.

[44] J. Smith, A. Gounaris, P. Watson, N.W. Paton, A.A.A. Fernandes, and R. Sakellariou. Distributed query processing on the grid. *Intl. J. High Performance Computing Applications*, 17(4):353–368, 2003.

[45] J. Smith, S. Sampaio, P. Watson, and N.W. Paton. The polar parallel object database server. *Distributed and Parallel Databases*, 16(3):275–319, 2004.

[46] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *Int. Conference on Very Large Data Bases VLDB*, pages 355–366, 2006.

[47] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB J.*, 5(1):48–63, 1996.

[48] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor – a distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, 2002.

[49] D. Thain, T. Tannenbaum, and M. 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.

[50] Y. Theodoridis. Ten benchmark queries for location-based services. *The Computer Journal*, 46(6):713–725, 2003.

[51] A. N. Wilschut, J. Flokstra, and P. M. G. Apers. Parallelism in a main-memory DBMS: The performance of prisma/db. In *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings.*, pages 521–532. Morgan Kaufmann, 1992.

[52] M.F. Worboys. A unified model for spatial and temporal information. *The Computer Journal*, 37(1):26–34, 1994.

[53] X. Xiong, H.G. Elmongui, X. Chai, and W.G. Aref. Place*: a distributed spatio-temporal data stream management system for moving objects. In *Proceedings of the International Conference on Mobile Data Management*, pages 44–51, 2007.

[54] S. Zanikolas and R. Sakellariou. A taxonomy of grid monitoring systems. *Future Generation Comp. Syst.*, 21(1):163–188, 2005.

[55] D. Zeinalipour-Yazti, S. Lin, and D. Gunopoulos. Distributed spatio-temporal similarity search. In *Proceedings of the International Conference on Information and Knowledge Management*, pages 14–23, 2006.