DATA INTEGRATION OVER HORIZONTALLY PARTITIONED DATABASES IN SERVICE-ORIENTED DATA GRIDS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

HATİCE KEVSER SÖNMEZ SUNERCAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2010

Approval of the thesis:

DATA INTEGRATION OVER HORIZONTALLY PARTITIONED DATABASES IN SERVICE-ORIENTED DATA GRIDS

submitted by **HATİCE KEVSER SÖNMEZ SUNERCAN** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Adnan Yazıcı _____
Head of Department, **Computer Engineering**

Assoc. Prof. Dr. Fehime Nihan Kesim Çiçekli _____
Supervisor, **Computer Engineering Dept., METU**

Dr. Mahmut Nedim Alpdemir _____
Co-supervisor, **TÜBİTAK-UEKAE İLTAREN**

Examining Committee Members:

Prof. Dr. Müslim Bozyiğit _____
Computer Engineering Dept., METU

Assoc. Prof. Dr. Fehime Nihan Kesim Çiçekli _____
Computer Engineering Dept., METU

Assoc. Prof. Dr. Ahmet Coşar _____
Computer Engineering Dept., METU

Dr. Cevat Şener _____
Computer Engineering Dept., METU

Ahmet Murat Özdemiray _____
Senior Researcher, TÜBİTAK-UEKAE İLTAREN

Date: **02.09.2010**

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: HATİCE KEVSER SÖNMEZ SUNERCAN

Signature :

ABSTRACT

DATA INTEGRATION OVER HORIZONTALLY PARTITIONED DATABASES
IN SERVICE-ORIENTED DATA GRIDS

Sunercan, Hatice Kevser Sönmez
M.S., Department of Computer Engineering
Supervisor : Assoc. Prof. Dr. Fehime Nihan Kesim Çiçekli
Co-Supervisor : Dr. Mahmut Nedim Alpdemir

September 2010, 78 pages

Information integration over distributed and heterogeneous resources has been challenging in many terms: coping with various kinds of heterogeneity including data model, platform, access interfaces; coping with various forms of data distribution and maintenance policies, scalability, performance, security and trust, reliability and resilience, legal issues etc. It is obvious that each of these dimensions deserves a separate thread of research efforts. One particular challenge among the ones listed above that is more relevant to the work presented in this thesis is coping with various forms of data distribution and maintenance policies.

This thesis aims to provide a service-oriented data integration solution over data Grids for cases where distributed data sources are partitioned with overlapping sections of various proportions. This is an interesting variation which combines both replicated and partitioned data within the same data management framework. Thus, the data management infrastructure has to deal with specific challenges regarding the identification, access and aggregation of partitioned data with varying proportions of overlapping sections. To provide a solution we have extended OGSA-DAI DQP, a well-known service-oriented data access and integration

middleware with distributed query processing facilities, by incorporating *UnionPartitions* operator into its algebra in order to cope with various unusual forms of horizontally partitioned databases. As a result; our solution extends OGSA-DAI DQP, in two points; 1 - A new operator type is added to the algebra to perform a specialized union of the partitions with different characteristics, 2 - OGSA-DAI DQP Federation Description is extended to include some more metadata to facilitate the successful execution of the newly introduced operator.

Keywords: Distributed Query Processing, Data Partitioning, Service-Oriented Data Grids, Data Integration

ÖZ

SERVİS YÖNELİMLİ VERİ IZGARA ORTAMINDA YATAY BÖLÜNMÜŞ VERİTABANLARI ÜZERİNDE VERİ ENTEGRASYONU

Sunercan, Hatice Kevser Sönmez
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Fehime Nihan Kesim Çiçekli
Ortak Tez Yöneticisi : Dr. Mahmut Nedim Alpdemir

Eylül 2010, 78 sayfa

Dağıtık ve heterojen kaynaklar üzerinde veri entegrasyonu bir takım zorlayıcı etkenleri de beraberinde getirmektedir. Bunlara örnek olarak; veri modeli, platform, erişim arayüzleri gibi çeşitli çoktürellik durumları, muhtelif veri dağıtım şekilleri, farklı bakım poliçeleri, ölçeklenebilirlik, performans, güvenlik ve güvenilirlik, dayanıklılık, yasallık vb. gibi durumların uygun yönetilmesi verilebilir. Belirtilen etkenlerden, farklı veri dağıtım ve bakım poliçelerinin yönetimlerinin bu çalışma ile en ilgili olduğu değerlendirilmektedir.

Tez kapsamında, veri kaynaklarının farklı oranlarda örtüşen dilimler içerecek şekilde bölünerek dağıtıldığı veri ızgaraları için, servis-yönelimli veri entegrasyon çözümü aranmaktadır. Bu ve benzeri ortamlar birebir kopyalanmış ya da bölünmüş verileri aynı veri yönetim çatısı altında toplaması açısından ilgi çekici bir çeşitliliktir. Bu çeşitliliği kullanabilmek için, veri yönetim altyapısı; farklı oranlarda örtüşen dilimli, bölünmüş veritabanlarında kimlikleştirme, erişim ve yığılma gibi özgün sorunlara çözüm üretebilmelidir. Belirtilen duruma çözüm üretmek adına; yatay bölünmüş veritabanlarının alışılmamış durumlarıyla başa çıkmak için, iyi bilinen, servis-yönelimli, dağıtık sorgu işleme özellikli, veri erişim ve entegrasyon ara

yazılımı olan OGSA-DAI DQP çatısının cebirsel yapısına; yeni bir operatör, UnionPartitions, eklenmiştir. Sonuç olarak OGSA-DAI DQP çatısı iki yönüyle genişletilmiştir; 1-Bölünmüş veritabanlarının farklı özelliklerdeki parçalarının özelleşmiş bir operatör ile birleşiminin sağlanması için yeni bir operatör tipi eklenmesi, 2- OGSA-DAI DQP federasyon tanımlamasının yeni tanıtılan operatörün çalışmasını kolaylaştırmak için daha fazla üstveri yönetecek şekilde geniş- letilmesi.

Anahtar Kelimeler: Dağıtık Sorgu İşleme, Veri Bölümleme, Servis Yönelimli Veri Izgaraları, Veri Entegrasyonu

To my husband, Ömer...

ACKNOWLEDGMENTS

I am deeply grateful to my supervisor Assoc. Prof. Dr. Fehime Nihan Kesim Çiçekli and co-supervisor Dr. Mahmut Nedim Alpdemir, for their contribution to my education, showing me the directions to follow in this study, giving their time and help constantly, and especially for motivating me since the very beginning of my study.

I would like to thank to State Planning Organization under the Office of Prime Ministry of Turkish Government and The Scientific and Technological Research Council of Turkey (TÜBİTAK) for supporting the study.

I would like to thank my colleagues in TÜBİTAK-UEKAE İLTAREN Unit, who always supported me and shared their experiences liberally. I would also like to thank to Gülşah Karaduman and Doruk Bozağaç for sharing indispensable machines and test running schedule.

I would like to thank to my parents-in-law for their sincere affection and bringing up such a tactful son and my sisters-in-laws for their geniality and understanding. Without their warmth and familiarity; this study would not have been completed.

I would like to thank my parents for all their love, reinforcement, teachings and courage throughout my whole life. I would like to thank my sister, Tuba, who has been one step ahead of me to light my way; my brother, Ömer, who made life less obsessed and my little sister, Ayşenur, who always had something to make me cheer. Without love and self-sacrifice of my family, I should not have been the person I am now.

Finally; I owe my deepest thanks to my dear husband Ömer, for making his best to help me study in every occasion and for insisting that I ignore him and study. His endless love, patience, experiences, support and surprises always motivated me during this thesis.

TABLE OF CONTENTS

ABSTRACT iv

ÖZ vi

ACKNOWLEDGMENTS ix

TABLE OF CONTENTS x

LIST OF TABLES xii

LIST OF FIGURES xiii

CHAPTERS

1 INTRODUCTION 1

1.1 Overview and Motivation 1

1.2 Contributions 4

1.3 Thesis Organization 5

2 BACKGROUND INFORMATION AND RELATED WORK 6

2.1 Distributed Query Processing 6

2.2 Distributed Query Processing in Service Oriented Data Grids 11

2.2.1 Open Grid Services Architecture - Data Access and Integration (OGSA-DAI) 13

2.2.2 Open Grid Services Architecture - Distributed Query Processing (OGSA-DQP) 14

2.2.3 Open Grid Services Architecture - Data Access and Integration - Distributed Query Processing (OGSA DAI DQP) 17

2.3 Database Partitioning 18

2.4 Multi-Node Horizontal Partitioning in Distributed Environments . . 21

3 EXTENDING OGSA-DAI DQP TO SUPPORT PARTITIONED DATABASES 26

3.1 General Considerations on the Extension 26

3.2 Extending OGSA-DAI DQP 29
3.2.1 Extending OGSA-DAI DQP Federation Description 29
3.2.2 Extending the Optimization Chain and Activities 30
3.3 New Operator Design 32
4 THE DESIGN AND IMPLEMENTATION OF THE *UNIONPARTITIONS* OPERATOR 34
4.1 *UnionPartitions* as a Binary Operator 34
4.1.1 Locating the Binary *UnionPartitions* Operator in Query Plan Tree 35
4.1.2 The Execution of the Binary *UnionPartitions* Operator . . 42
4.2 *UnionPartitions* as an N-ary Operator 45
4.2.1 Locating N-ary *UnionPartitions* Operator Strictly in the Query Plan Tree 45
4.2.2 Locating the N-ary and Binary *UnionPartitions* Operators in the Same Query Plan Tree 45
4.2.3 The Execution of the N-ary *UnionPartitions* Operator . . . 47
5 EXPERIMENTS and EVALUATIONS 49
5.1 Data Preparation 49
5.2 Test Environment Setup 50
5.3 Experiment Design 52
5.3.1 Effect of Data Size on Performance 54
5.3.2 Effect of Increasing Overlapping Sections Among Partitions 55
5.3.3 Effect of Increasing Operator Fan-In 56
5.3.4 Comparison of Binary and Hybrid Trees 58
5.3.5 Extension Scalability 59
6 CONCLUSIONS AND FUTURE WORK 63
REFERENCES 66
APPENDICES
A TEST DATA GENERATION FILE 72
B DQP TEST MANAGER 75

LIST OF TABLES

TABLES

Table 4.1 Example Partition Information 39
Table 4.2 Initial Partition Matrix 39
Table 4.3 Updated Partition Matrix - 1 40
Table 4.4 Updated Partition Matrix - 2 41
Table 4.5 Updated Partition Matrix - 3 41
Table 4.6 Updated Partition Matrix - 4 42

Table 5.1 Schema for Users Tables 50
Table 5.2 Schema for Projects Tables 50
Table 5.3 Schema for Allocations Tables 50
Table 5.4 Execution Times for Experiment 5.3.1 55
Table 5.5 Queries Used in Experiments; 5.3.2, 5.3.3 and 5.3.4 56
Table 5.6 Cardinality of Partitions when 50% Overlapping 58
Table 5.7 Updated Schema for Users Tables 61

LIST OF FIGURES

FIGURES

Figure 2.1 Query Processing Steps [1] 7

Figure 2.2 OGSA-DAI Architecture [2] 15

Figure 2.3 DQP Query Compiler Execution Interactions [3] 16

Figure 2.4 The use of horizontal partitioning along the database evolution [4] 24

Figure 3.1 Example Partition Information 30

Figure 3.2 Default Optimization Chain of OGSA-DAI DQP 31

Figure 4.1 Pseudocode for Ordering Binary UnionPartitions Operator 37

Figure 4.2 An example to UnionPartitions Operator Tree 43

Figure 4.3 Pseudocode for Execution of Binary UnionPartitions Operator 44

Figure 4.4 An Example to UnionPartitionsNary Operator Tree 46

Figure 4.5 An Example to UnionPartitions Operator Tree with Hybrid Approach . . . 47

Figure 5.1 DQP Test Manager Main View 53

Figure 5.2 Queries Used in Experiment 5.3.1 54

Figure 5.3 Query Execution Results For Experiment 5.3.1 55

Figure 5.4 Query Execution Results For Experiment 5.3.2 57

Figure 5.5 Query Execution Results For Experiment 5.3.3 59

Figure 5.6 Query Execution Results For Experiment 5.3.4 60

Figure 5.7 Query Used in Experiment 5.3.5 60

Figure 5.8 Query Execution Results For Experiment 5.3.5 62

Figure B.1 Example Query Execution 75

Figure B.2 Example Query Execution Result 76
Figure B.3 Example Query Execution Plan 76
Figure B.4 Example Query Execution Test File 77

CHAPTER 1

INTRODUCTION

1.1 Overview and Motivation

As the proliferation of information resources over the Internet has gained more and more momentum in recent years, information integration has become both more crucial and challenging for a growing range of applications that aim at providing an integrated view over distributed and heterogeneous resources. The challenge has many dimensions: coping with various kinds of heterogeneity including data model, platform, access interface heterogeneities, scalability, performance, security and trust, reliability and resilience, access control, auditing and accounting, legal issues etc. Each of these dimensions deserves a separate thread of research efforts; and indeed they did draw sufficient attention to cause the culmination of a considerable amount of literature. One particular challenge among the ones listed above that is more relevant to the work presented in this thesis is coping with various forms of data distribution and maintenance policies.

Looking from an architectural point of view, information integration infrastructures follow different approaches in how they treat the data sources and in how they produce the integrated forms of data [1], [5]. Data warehousing approach[6], for instance, designates a single information store for the integrated data and relies on periodic updates of that central store from the source databases via pre-defined data transformation filters. As such, the data warehousing approach prioritizes high-performance requirements over other requirements such as data ownership, high update frequency of source databases, flexibility etc. Mediator-Wrapper approach [7], on the other hand, offers a dynamic data integration scheme where source databases are accessed just before the integration via a standard wrapper and the in-

tegration process takes place on-the fly. As such, the Mediator-Wrapper approach aims at meeting the requirements of a class of applications where the preservation of data ownership, just-in-time integration and flexibility are essential [7]. The latter approach is of particular interest for the work presented in this thesis.

In a typical data integration scenario over the internet, data source hosting and data provision enterprise spans multiple administrative domains. The relationships between those data sources and methodologies to access and integrate data over them are not always straight forward. For instance, the distributed data sources may be replicas or partitions of the same database, rather than being logically distinct but related data segments with different schemas. The case where the distributed data sources have replicated or partitioned data sections is of particular importance for the study presented in this thesis. To be more specific, the problem we have set out to solve was to handle cases where a database was distributed to multiple independent administrative domains with the same data content initially (i.e. as replicas), but where parts of those multiple copies evolved into overlapping partitions over time through independent data insertions carried out within each administrative domain. In this scenario the replicas are not generated to support more timely query execution, but rather as a consequence of administrative policies. This may not be a particularly common case for distributed database applications; however it is a practical requirement for our target application area where a set of pre-defined simulation scenarios are distributed to multiple institutions and users create new scenarios to their local set causing the local set to grow. This effectively results in partitioned data with overlapping sections of various proportions (due to the initial replication process).

Database partitioning is a relatively advanced topic in data management area, mostly to support applications that require high performance or high availability for large volumes of data [8, 9, 10]. Mainstream commercial Database Management System (DBMS) vendors provide solutions for various kinds of partitioning techniques such as range partitioning, list partitioning, hash partitioning or a combination of those. The selection of the appropriate technique would depend on the characteristics of the data or the primary purpose of the partitioning. Nevertheless, the available solutions are vendor dependent and normally require tightly controlled operating environments both from an administrative point of view and from a run-time execution constraints point of view. The work presented here aims at adding the capability of handling partitioned data to a service oriented data integration middleware, in a Grid envi-

ronment where a more liberal operating environment is assumed for both data resources and computational resources involved.

Service-oriented data integration middleware that operate in a Grid environment are still in their infancy due to the requirement to cope with many challenges introduced by the characteristics of the environment, as mentioned earlier in this chapter. One of the fundamental challenges in a data integration context, for instance, is to cope with different modes of heterogeneities such as data model heterogeneity. Data to be integrated can exist in many forms including object-oriented, relational or XML databases, or even in plain text files on the disk of a computer. Service-oriented middleware are promising in their ability to compensate for such heterogeneities by providing software layers that offer standard access interfaces and protocols of interaction, leading to a flexible programming model. In fact, service oriented architectures have been in the focus of many research activities that aim at tackling not only the data model heterogeneity but other challenges mentioned above. One architectural specification that is of particular importance in the work presented in this thesis is the Open Grid Services Architecture (OGSA) [11]. OGSA defines the components, interfaces and the functionality required to implement a service-oriented Grid. As such, it specifies the properties of the foundation layer in a multi-tier software infrastructure required to harness the resources made available on the Grid through well-defined and uniform mechanisms [11]. Building on that foundation OGSA-Data Access and Integration (DAI) - Distributed Query Processing (DQP) has two main dimensions;

- Specifying standard interfaces for a Grid Data Service (GDS) to facilitate the construction of a platform independent and data model independent data access layer on the Grid [2].
- Providing a higher-level middleware layer that supports queries over multiple grid data services combining data access with access to analysis services in a single framework [12].

DQP dimension in OGSA-DAI DQP has come to posses two distinct properties [12]:

1. It supports low-cost data integration, in that it uses existing OGSA-DAI wrappers to obtain access to networked resources, and in that there is no need to map source schemas to a single global model.

2. It builds on parallel database technology, in which both pipelined and independent parallelism are used to generate initial results early and to increase throughput.

To maximize the benefit of the latter property, it is essential to exploit replicated and/or partitioned data if available in the environment. With these properties, DQP dimension in OGSA-DAI DQP provides the necessary framework for meeting our specific requirements mentioned earlier (i.e. the case where a certain database was distributed to multiple administrative domains with the same data content initially (i.e. as replicas), but where parts of those multiple copies evolved into overlapping partitions in time through independent data insertions). However, as it stands, OGSA-DAI DQP did not have inherent support specialized to partitioned or replicated databases. This thesis study aims to provide a solution to the management of various unusual forms of horizontally partitioned databases.

1.2 Contributions

This thesis extends OGSA-DAI DQP to allow for the data resources to include horizontally partitioned databases in such a way that access and integration logic required to handle the partitioned data remains transparent to the query constructor (i.e. to the daily user who poses data integration queries using OGSA-DQP). The extensions are essentially in two points;

- A new operator type is added to the algebra to perform a specialized union of the partitions with different characteristics.

- OGSA-DAI DQP Federation Description is extended to include some more metadata to facilitate the successful execution of the newly introduced operator.

Finally, we report our findings on the performance behavior of queries using the extensions. Since the extensions are non-disruptive to the architecture and to the fundamental run-time execution mechanisms of OGSA-DAI DQP, and since the overall run-time characteristics of OGSA-DQP are already well-documented [12, 13], our performance experiments focus only on the impact of our extensions.

1.3 Thesis Organization

This thesis is organized as follows: In Chapter 2, the work in literature related to the issues to be facilitated to solve the marked problem (also possible sub-problems) in this thesis together with an overview of the software architectures extended by this study is presented. Chapter 3 gives a more detailed picture of the problem and explains the preparations for the presented solution. In Chapter 4; the solution proposed by this thesis to the target problem is given and a new operator is presented. Chapter 5 contains the experimental results for the newly proposed operator with sample test cases. Finally, we conclude the thesis and point to the possible future extensions in Chapter 6.

CHAPTER 2

BACKGROUND INFORMATION AND RELATED WORK

2.1 Distributed Query Processing

The fact that the increasing number of Grid applications make use of huge amounts of data, has led to the generation of middleware applications with high-level data management functionalities like data derivation [14], replication [15], database access and management [16], resource monitoring, discovery, management, security [17, 18], generic data access services between client and DBMSs; OGSA-DAI [2], Spitfire [19] and query processing; OGSA-DQP [12]. Distributed Query Processing is one of the main functionalities that provides seamless and simultaneous read accesses to several databases which are possibly located on shared-nothing computational systems. Systems with DQP facilities allow their clients to ask for queries involving different resources at different sites. Although it may vary from system to system, a typical DQP system provides the following functionalities [20]:

- *Schema Integration:* The schemas in each resource are integrated to form a global schema for the large-scale view of the system-wide resources.
- *Location Transparency:* The clients are not bothered with the location of the resources they are using. The queries are automatically directed to the locations of required databases.
- *Table Partitioning:* A table can be decomposed into smaller parts and distributed over the network. The system automatically handles table partitioning without bothering the clients about the partitions and their locations.
- *Multiple Vendor Support:* A DQP system should be able to use resources from different

vendors, in a seamless way.

- *Multiple Data Format Support:* A DQP system should support clients to perform SQL joins not-only on relational resources but also non-relational resources.
- *Multiple Site Support:* As the name implies, DQP allows the data to be delivered to several resources.
- *Administration Facilities:* A reliable DQP system should also support its users with global security and audit trails.

Supporting all the functionalities above is not a trivial task. So, the literature contains a wide range of distributed query processing studies.

Figure 2.1 shows the architecture of a general query processing engine that is enhanced in both centralized and distributed environments.

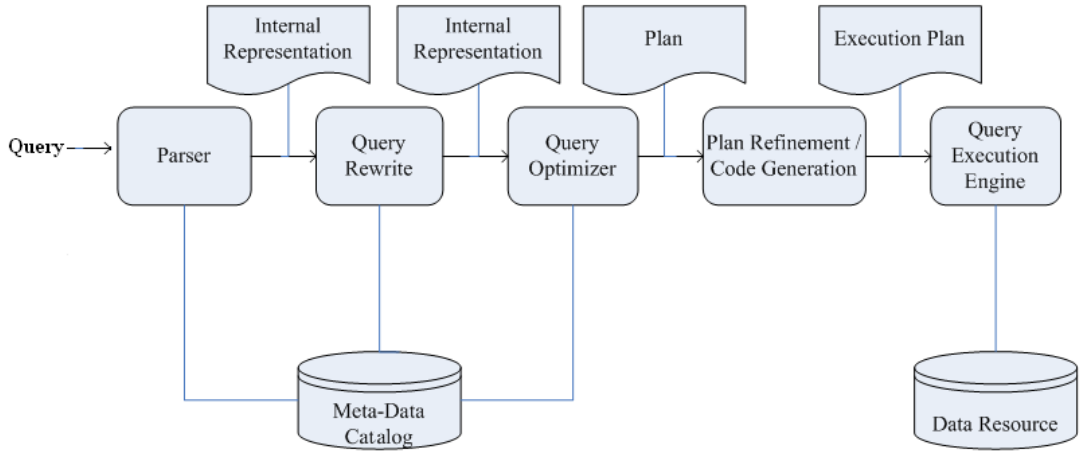

Figure 2.1: Query Processing Steps [1]

The input query to a query processing engine is usually in the form of SQL or OQL like query languages. Upon receiving a query request from a client; first the parser module parses the query and then converts it to an internal representation to be used throughout the query execution engine. Secondly; the query rewrite module takes the internal representation to optimize the query in terms of shape (i.e. written form) like eliminating redundant predicates, simplifying expressions and disclosing views. The optimizations related to the physical properties

of the systems are applied by the next module, query optimizer. The optimizer module is mainly responsible for producing a query plan where the operators and the application order of operators are defined. Query plans are generally represented as trees with nodes as operators and edges as communication links indicating the consumer/producer relationships. Each node is responsible for carrying out one activity. In addition to producing a query plan, the query optimizer also decides on the execution strategies of operators and the assignment of operators to sites for execution in distributed environments. The query optimizer bases all these decisions on a cost model, and selects the one with the low cost from alternative plans as the output plan. As a fourth step, the plan refinement/query execution module takes in the produced query plan to produce an executable query plan with transformations of expression evaluations and predicates. In some systems, this plan refinement module is combined with the query optimizer module. Finally, the query execution engine, the component providing the implementation of every operator; comes into scene to handle the evaluation of the query according to the produced query execution plan. In general, engines are based on the iterator model [21], where each operator is implemented as iterators so that they have the same interface. This model allows for connecting operators as consumers/producers one after another and enhances pipelining of results from producer operators to consumer operators without waiting for the completion of the execution of producer operators. During the phases of parse, rewrite or optimization steps, the required information is provided from the catalog. The database schema (e.g. table definitions, table views etc.), the partitioning schema (e.g. partitioning information of tables) and physical information of resources (e.g. indices) are the main elements of a catalog [1]. The literature contains several DQP systems using either complete, partial or extended versions of these query processing steps.

Polar*, which is not service-based, is accepted to be the first study to employ distributed query processing in a Grid environment [12]. It exploits Polar system, which is a parallel object database server running on a shared-nothing parallel machine, with facilities of DQP by using MPICH-G [MPI] (i.e. grid enabled MPI). The query compiler consists of the parser, logical optimizer, physical optimizer, partitioner, scheduler and evaluator modules; and is responsible for the generation and evaluation of efficient query execution plans for OQL queries with two-step optimization paradigm in a distributed setting [22]. Pursuing the operator model of parallelization, Polar*, enhances the use of exchange operators at locations where data input needs repartitioning in order to be processed on multiple nodes. For a distributed environment,

the exchange operators in the query plan also indicate the sub-plans that will be executed on separate nodes. The evaluator of Polar* is based on the iterator model [21], connecting operators with interface functions of *open()*, *next()* and *close()*.

There have been several other studies like GridDB [23], GridDB-Lite [24] and POQSEC [25]; for pre-service-based grids enhancing database query languages to define application requirements that will be processed over Grid middleware. GridDB [23] allows users to access and manage process-centric grid services and use a DML to manage the parameters and results of grid computations by providing a data-centric overlay upon process-centric middleware. GridDB enables computational steering in order to change the batch-oriented behavior of process-centric middleware to be able to access partial results [23]. GridDB-Lite (also called STORM); mainly presents two tasks; extracting data effectively from distributed databases and transferring that data from storages to evaluation nodes [24]. In executing queries, GridDB-Lite uses the inspector/executor [26] model; where the inspector is responsible for generating a schedule of data movement from storage sites to processors and the executor is the one to apply that schedule and perform data movements. Upon getting a query request;

1. Indexing is enhanced to determine the tuples to be gathered from each data source by using *data source service* which provides a virtual dataset view of tables. Tuples are categorized as select attributes (i.e. attributes satisfying select predicates), partition attributes (i.e. attributes deciding on the partitioning of result among processors) and result attributes (i.e. attributes returned as part of result) *[Inpector Phase]*

2. An unfiltered planning table is generated by using *data source service*. The table includes a unique ID for each tuple, select attribute values and partition attributes values. *[Inpector Phase]*

3. This unfiltered planning table is converted to a filtered planning table by conducting *filtering service* for the removal of tuples not satisfying the select attributes. *[Inpector Phase]*

4. The filtered planning table is then passed to *partition generation service* to determine the result tuple partitioning over processors. *[Inpector Phase]*

5. Partitioning information and filtering information are passed to *data mover service* to

compute communications schedule and perform data movement. *[Executor Phase]*

Although GridDB-Lite uses services in execution and it is actually service based, it is not web services based; so the standard service interfaces are not used.

POQSEC *(Parallel Object Query System for Expensive Computations)* system [25]; deals with the management of long-running queries that use a lot of computational resource in the grid. The system takes SQL-like queries expressed in an application-oriented schema and converts them to job descriptions that can be submitted for execution. By this way, the user is saved from the complexity of details of job descriptions and their parallelization [25]. It makes use of computational and data resources of pre-existing grid infrastructures; [27, 28], respectively. This puts some restrictions on POQSEC, like full control of computational resources and possibilities of acquiring them that differentiate the system from stand-alone generic distributed query processors discarding such anomalies. POQSEC consists of four components: query coordinators, start-up drivers, executors and supervisors [25]. Upon getting a query request;

1. *Query Coordinator* first checks, whether further computational resource is required. If not, the request is answered using the local database; else it separates the query into sub-queries and submits to NorduGrid for parallel execution by providing the required information (e.g. xRSL job scripts, computational resources requirements, startup procedures for computing elements and input/output of each sub-query).

2. In the *computing elements (CE)* of NorduGrid, *start-up driver* starts POQSEC modules for execution initialization and supervisor selection. After then, *start-up drivers* start *executors*.

3. *Executors* first subscribe to the selected supervisor and then execute the query on data available at that CE. By the help of *supervisors*, the *query coordinator* follows the execution of the submitted queries on *computing elements* (CE) available through NorduGrid infrastructure [29].

For more information on additional distributed query processing arguments like optimization strategies (adaptive, dynamic and parallel approaches), roadmaps in different network models (e.g. server-client, peer-to-peer and multi-teer), execution strategies, execution engine models etc., the reader is encouraged to refer to [1]. Due to concern of this thesis; the survey is

in general, narrowed to distributed query processing functionalities enhanced in a service-oriented grid setting that will be detailed in the next section.

2.2 Distributed Query Processing in Service Oriented Data Grids

Ease of use, loose coupling, reusability, interoperability with applications, seamless integration and platform independence properties of web services and in turn a data grid infrastructure providing these; suits the needs of a distributed heterogeneous environment with different applications, platforms, languages and databases. Service-oriented data grids support distributed-data-intensive applications with services of resource discovery, data transport, management, modification and dataset replication over sites [30]. Our concern in this thesis is on service-oriented data grids that are using the well-known web services paradigm as the underlying infrastructure. The need for a service interface format for a platform independent data grid environment that separates implementation details from service interfaces by web-services paradigm has led to the emergence of architectural specification OGSA (Open Grid Services Architecture) [11]. OGSA-DAI and OGSA-DQP, and after coalition of the two, OGSA-DAI DQP; which are the base for this study (to be detailed in the following subsections) are well-known compliant of OGSA specification.

In addition to OGSA-DAI DQP system, there are several other service-based systems targeting DQP facilities like; Web Service Management System (WSMS) [31], SkyQuery [32], Garlic [33] and Kleisli [34]. WSMS [31] allows clients to query multiple web services simultaneously using an SQL-like interface in a seamless and integrated manner while benefiting from pipelined parallelism. [31] defines a WSMS architecture to contain three components;

- *Metadata Component:* Responsible for the management of catalog data, registration of web services and providing web services schemas
- *Query Processing and Optimization Component:* Responsible for the management of the execution of queries by producing optimized query plans using the information provided by the profiling and statistics component.
- *Profiling and Statistics Component:* Responsible for profiling web services characteristics like; response times, data statistics of web service data.

In executing queries; WSMS benefit from pipelining parallelism by chaining web services to form workflows where processed data is streamed through one web service to another. In producing the query execution pipeline; the query processing and optimizer module considers issues like precedence constraints between services, response time of each service, chunk size of delivered data, etc. The queries handled by [31] are simple select-project-join queries over web services by considering web services as virtual relations.

Although; SkyQuery [32] lacks generality and targets mainly astronomical federation data, it is still a good application of service-oriented DQP. It uses a mediator-wrapper approach and contains three components; Clients, Portal and SkyNodes. The Clients are the ones to meet queries and they are responsible for passing the query requests to the Portal. Portal being the mediator interposes between the Clients and the SkyNodes. SkyNodes, implementing wrappers over DBMSs, join to federations by using the Registration service of the Portal. Upon receiving the query from Client, Portal generates performance queries by decomposing the query and then using the results of these performance queries, creates an optimized execution plan to be sent to SkyNodes for evaluation. The plan contains a chain of queries ordered according to execution turn among SkyNodes. During execution of the plan from one SkyNode to other, partial results are not sent to Portal only the final result is sent to the Portal. A SkyNode has four Web services interfaces; information service (i.e. provides astronomical constants of that SkyNode), meta-data service (i.e. provides schema information of DB), query service (i.e. queries database) and cross match service (i.e. execute astronomy specific cross-match queries) [32].

Garlic [33] and Kleisli [34] aims to support querying multiple distributed heterogeneous data resources in a declarative manner using wrapper like modules over data resources. Upon receiving an SQL query; *federated server* in Garlic [33] works with *wrapper components* collaboratively to form an execution plan (i.e. a decomposition of the query into fragments) using Garlic's *request-reply-compensate* protocol. At *request* stage; the federated server asks wrappers for the part of the query they can execute. After taking requests, each wrapper answers by a *reply* the part that can be executed and the required information for execution. And finally, receiving the replies, the federated server *compensates* for an execution plan. After selecting an execution plan; the query fragments are sent to wrappers for execution. After the completion of execution, each wrapper sends the results back to federated server which combines results coming from all wrappers [33]. Kleisli resembles Garlic in using *wrapper*

like modules named *data drivers* that deal with the specifics of various data resources. OQL queries accepted by Kleisli are first decomposed and then transformed for understanding of data drivers that will lead the execution of sub-queries by the help of *query optimization module* [34].

In the internet-scale query processing area; ObjectGlobe [35] takes interest by its similarity of goals with OGSA-DQP system. ObjectGlobe favors moving the execution logic close to data rather than moving whole data close to execution nodes. Being internet-wide scale, and non-service-oriented, ObjectGlobe holds records of the query operators, additional evaluation nodes and registries of data resources. The concepts and functionalities of ObjectGlobe, are seen in a more generalized and service-based manner in OGSA-DQP.

This thesis is based on OGSA DAI DQP system; that is chosen for its generality, ease applicability, success in several projects (i.e. ADMIRE[1], BIRN[2], GEO Grid[3], MESSAGE[4], BEinGRID[5], LaQuAT[6], Database Grid[7], etc. [36]), SQL support, compactness, community support and extendibility to be used for the purposes of this thesis.

In the following; a summary of each middleware, OGSA-DAI and OGSA-DQP, is presented separately first. Then the variations in the new coalition OGSA-DAI DQP are explained.

2.2.1 Open Grid Services Architecture - Data Access and Integration (OGSA-DAI)

OGSA defines the capabilities for the functionality required to implement service-oriented grid architecture by addressing the challenges of integration of services from distributed, heterogeneous and dynamic grid environments using Web services concepts and related technologies [11]. OGSA-DAI initiative on the other hand aims at making structured heterogeneous data (i.e. files, relational and XML databases) available to consumers on the grid, via standard access interfaces in a service oriented setting. The data resources are wrapped via grid service interfaces as defined in OGSA specification so that they can be accessible in a standard way using Web Services technologies such as WSDL and SOAP. While allowing

[1] http://www.admire-project.eu/
[2] http://www.birncommunity.org/
[3] http://www.geogrid.org/en/index.html
[4] http://bioinf.ncl.ac.uk/message/
[5] http://www.beingrid.eu/
[6] http://laquat.cerch.kcl.ac.uk/
[7] http://wiki.dbgrid.org/index.php

standard data access to heterogeneous data resources, OGSA-DAI also aims to support the movement of processing logic near data rather than moving large volumes of data near processing logic, to provide a convenient framework for grid-centric applications.

It is important to emphasize the central characteristics of OGSA-DAI compliant Grid Data Services to indicate the underlying programming model used by OGSA-DAI and OGSA-DQP. The interaction with a Grid Data Service is performed via a document oriented interface, where the request is specified by an XML document called the *perform document*. A perform document includes a sequence of *activities*. Activities represent well-defined tasks to be completed at the server side and are the main behavioural building blocks of OGSA-DAI Grid Data Service. The sequence of activities in the perform document specify an execution model on the server side where activities are linked one after the other to form a workflow. The workflows are also a way of optimization for OGSA-DAI service communications by reducing the amount of SOAP level data transfers, which is known to be relatively inefficient due to infrastructure and formatting overheads (e.g. verbose XML encoding, marshaling / unmarshaling etc.). Querying, transformation, integration and data delivery are examples of the tasks that can be achieved via activities. The activities are also extensible to include more tasks for uses of different applications with different requirements. Simple intermediary, persistent intermediary, redirector, coordinator and network assembly are the five different common use scenarios of OGSA-DAI [37]. Figure 2.2 summarizes the architecture of OGSA-DAI in a schematic way.

2.2.2 Open Grid Services Architecture - Distributed Query Processing (OGSA-DQP)

OGSA-DQP is service-based, fully compliant to the externally visible interfaces of a standard Grid Data Service since it is developed using the OGSA-DAI's extensibility points. By using OGSA-DAI as a base, it benefits from homogeneous access to heterogeneous resources and offers an opportunity to be used as a high-level on-the-fly data integration service. OGSA-DQP supports SQL queries over distributed relational database resources and encapsulates compilation, optimization and evaluation (execution) of those queries inside the processing logic of constituent services [12].

Through the higher level layering of its constituent services over OGSA-DAI Data Services,

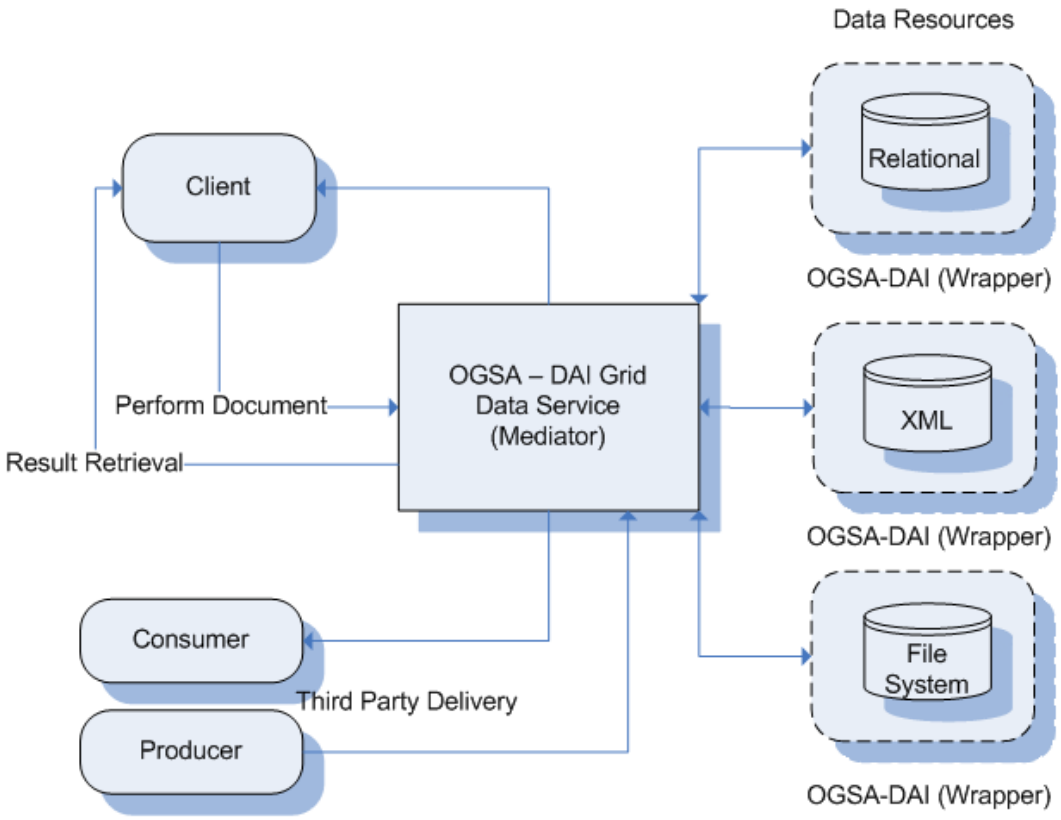

Figure 2.2: OGSA-DAI Architecture [2]

OGSA-DQP implements wrapper-mediator architecture; in that it acts as a mediator over data sources wrapped with OGSA-DAI services. OGSA-DAI is extended and two services are created to produce OGSA-DQP; coordinator and evaluator. The Coordinator service is the main entry point for the functionalities provided by OGSA-DQP, where a query is compiled, optimized and scheduled for evaluation by evaluator services. To provide the basis for a better understanding of our extensions to OGSA-DQP we present a summary of the main processing pipeline inside the Coordinator Service and its interaction with the Evaluator Services. For a more detailed account the reader is referred to [12, 13, 38, 39, 40].

Figure 2.3 illustrates a graphical representation of the sequence of the processing steps. Upon receiving an SQL query; OGSA-DQP first validates the query and then Parser parses the validated query to produce an *abstract syntax tree* (AST). After that *Type Checker* gets this AST to check the type elements of the tree and outputs an *annotated abstract syntax tree* (AAST) that contains the type information. Then, as the last step on the way to query plan building,

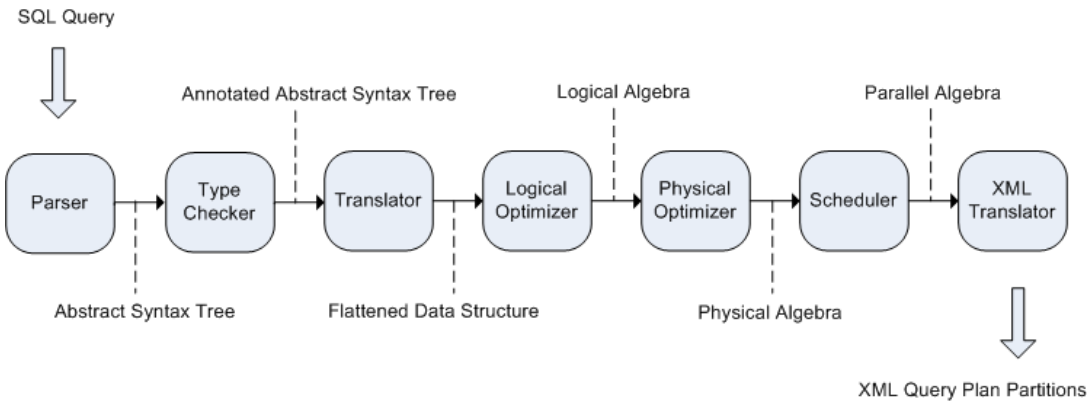

Figure 2.3: DQP Query Compiler Execution Interactions [3]

Translator takes AAST to produce an internal data structure (*flattened data structure*) to be used throughout the optimization.

After the conversion of SQL query to an internal data structure, creation of the query plan tree starts. Coordinator accomplishes this optimization in three blocks: *Logical Optimization*, *Physical Optimization* and *Parallel Optimization*. At the logical optimization step, a single non-distributed query plan is produced using the logical properties of data sources (e.g. table schema information). Upon taking the logical algebra produced, *Physical Optimizer* updates the plan according to the physical properties of data sources (e.g. indexing information, row size).

To exemplify, *Logical Optimizer* is the one to produce the join operation, whereas *Physical Optimizer* is the one to update the join operation to be a more specific join (e.g. *Theta-Join*, *HashJoin* or *HashNestedLoops*). After these optimization steps; the *Parallel Optimizer (Scheduler)* generates plan partitions while considering the parallelization over parallelizable operators (e.g. joins) and assigns these plan partitions to evaluators according to the properties of nodes (e.g. belonging networks). And then, as the last step to the evaluation, *XML Translator* converts these plan partitions to XML fragments.

The Coordinator Service sends each query plan partition to an Evaluator Service as specified by the scheduler. An Evaluator service includes the implementation logic for the execution of the query operators. Upon taking a fragment (or a partition) of the query execution tree from the coordinator service, an evaluator evaluates that query fragment by executing the algebraic

operators involved in it. The Evaluator services effectively form a coarse-grained execution tree where the root is the coordinator and each node is a query plan partition encapsulated by an evaluator service. As the query evaluation starts, the results of the query execution begin to flow bottom up as blocks of data tuples, from the leaf data sources, through the intermediate evaluator services up to the coordinator. In other words, a data flow machine dynamically forms results by acquiring and consuming resources during the query execution and is teared down at the end disposing the consumed resources.

2.2.3 Open Grid Services Architecture - Data Access and Integration - Distributed Query Processing (OGSA DAI DQP)

By release 3.2.1, OGSA-DQP is written from scratch and it is included in OGSA-DAI releases making deployment of DQP system much easier. The new version of the distributed query processor in OGSA-DAI still preserves the aims in its previous releases; but the architecture and the execution flow of queries are changed considerably.

Although, Coordinator service still exists and serves for the coordination of the execution steps of a requested query like, parsing, type checking, translating, optimizing and query plan partitioning; the management and the implementation of these steps have been changed. They are now more configurable, adaptable and extendible according to the specific project needs and environment [41]. Like the previous version; the schema of the virtual database is constructed before any query is executed. The considerable difference in the execution of a query is seen in the optimization step; where now an optimization chain is defined and given to the system as an XML configuration file. After the initial query plan is produced by the steps (e.g. parsing, type checking etc.) up to optimization; it is guided through that ordered optimization chain. Each element in the chain gets the query plan and produces a new optimized one according to its task. For example; the task for *ProjectPushDownOptimizer* is to find the project operators that can be pushed down in the query plan and then to update the query plan accordingly.

The partitioning is similar to the previous releases; each operator is assigned to an *Evaluation Node*, according to the needs of the query operator and resources available on the evaluation nodes. Separate evaluator services are removed and the evaluation logic of the query plan operators is located inside the whole system. That is; when the OGSA-DAI DQP system is

installed on a site; that site can be used freely as the coordinator, evaluator and wrapper for a data resource without any additional installation.

The execution logic of operators in query plans is adapted to activities that are widely used in OGSA-DAI. The optimized and evaluation node annotated query plan; is converted into multiple workflows (i.e. chain of activities) by the help of the builder classes of the corresponding operators. The corresponding executor activity of an operator in the query plan is given in the same file with the optimization chain; *DQPCompilerConfig.xml*. The workflows are then sent to the specified OGSA-DAI servers and executed in a parallel way [41], [42].

The study for this thesis started when OGSA-DAI and OGSA-DQP were two separate middleware. Therefore we made our implementation based on the architecture of separate OGSA-DQP middleware, initially. After their merge and combination to a single middleware; we updated the implementation to be in harmony with the new version. Being users that have developed applications using both data access and distributed query facilities of these two different versions; we can honestly say that the new version offers more friendly, understandable, extendible and adaptable interfaces.

2.3 Database Partitioning

A partition (fragment or segment) is a split of a database or its comprising elements (e.g. indexes etc.) into separate pieces. Each partition of the database may have its own storage characteristics independently from others. Partitioning; while providing administrators with the freedom to manage the partitions separately; gives a non-partitioned database view for the client side when working with SQL commands [8]. Partitioning is generally used for increasing manageability, scalability, performance, availability and security of a database. The mentioned characteristics of databases are affected from several reasons like; hardware strength (manageability, performance and scalability), database maintenance time, capturing database statistics, network failures (availability), effective database design and query rewrites (performance) etc. Partitioning data is one of the good strategies to enhance the encountered problems by possibly increasing the controllability of the mentioned characteristics.

- *Manageability;* refers to the control of data in the database and it can be taken as a measure of quality of handling generally space, memory and processor constraints in

databases [9]. The larger the database; the larger the likeliness of problems with these constraints occurring. For example; for an application producing data on a daily basis; when it is time to load data, the addition of a single partition is faster compared to the update of an entire table [8]. Another good example is backup operations of databases. When the backup of a portion of data is required, it is easier to backup the partitions containing the requested data rather than backing up the whole non-partitioned table.

- *Scalability;* refers to the ability of a database to continue its function without any performance penalty when the number of users or database size increases. By partitioning the database to several smaller parts; the total number of client requests to the database will also be divided over partitions.

- *Performance;* refers to the quality of services received from a database like; querying, updating or index creating. Parallelization of requests to a partitioned database may increase the performance that can be achieved by operating over different partitions simultaneously. It is also possible for some requests with queries on partitioning keys to eliminate the redundant processing of data on unrelated partitions.

- *Availability;* refers to the 7/24 active operation of databases. Partitioning not only divides data but also divides the possibility of failures emerging from a single table/database over several tables/databases. So; when a problem occurs at a partition; the queries unrelated with the failed partition would continue to be answered successfully. The term *five nines* is used to describe highly available systems; since hundred percent availability of a database is not accepted to be a reality [10].

- *Security;* the confidential parts of data are separated and stored at the required partitions only. This prevents the access of confidential data to unauthorized users.

Although partitioning can be used to accomplish a number of various objectives, the most common goal is to reduce the amount of data reads for particular SQL operations to reduce the overall response times. There are two major forms of partitioning;

Horizontal Partitioning The height of a table is divided horizontally such that; disjoint data sets with rows as elements are produced that can be accessed individually (one partition) or collectively (one-to-all partitions) [43]. All the attributes of the table exist in all the partitions. That is, schemas for all the partitions are exactly the same with

the original non-partitioned database. Horizontal partitioning has two sub-partitioning schemes; primary and derived. In primary partitioning, the partitioning attributes are selected from the attributes of the local relation (table), whereas derived partitioning uses predicates based on foreign relations (tables) in fragmenting the relation (table) in question. Derived partitioning is meaningful if and only if there is a relationship between the local and referenced relations [44]. Horizontal data partitioning is used as a major technique used in the design of data warehouses. An example for horizontal partitioning is dividing a sold items market database with 12 months of historical data, into 12 distinct partitions, where each partition contains the data for a single month. During the construction of horizontal partitions; the selection operator is used together with a partitioning scheme. The reconstruction of the non-partitioned database from partitions is possible with the union operator.

Vertical Partitioning The width of a table is divided into several partitions vertically; producing partitions with different columns but with all rows of the table. The exception to column differentiation is primary keys that exist in all the partitions to be used during the reconstruction of the non-partitioned database. An example vertical partitioning is the division of a large video database with large BLOB data into partitions to put frequently accessed columns on one side and rarely accessed BLOB data on the other side [43]. During the construction of vertical partitions, the projection operator is used together with a partitioning scheme. The reconstruction of the non-partitioned database from partitions is possible with the join operator.

The correctness of a partitioning technique is checked by the three rules: [45]

- Completeness; each data item in relation R; should also exist in at least one of the partitions of the relation R; R_1, R_2, ..., R_n.
- Reconstruction; there exists an operator that reconstructs the original relation from the fragments.
- Disjointness; each data item of relation R, occurs only in one relation. Tuple is the data item for horizontal partitions and attribute is the data item for vertically partitions. There is an exception for this rule; when vertical partitioning, is in question with the obligation of carrying the primary key columns to all partitions.

There are various studies on database partitioning with various issues like; physical partitioning techniques, allocation of partitions, replication of partitions, on-the-fly partitioning techniques for query processing, migration of data among partitions, etc. over both relational and object-oriented database management systems. Due to the concern of this thesis, the research is narrowed to distributed horizontal partitioning schemes of relational databases that are detailed in the following section.

2.4 Multi-Node Horizontal Partitioning in Distributed Environments

The mainstream commercial DBMS vendors provide solutions with data definition language (DDL) support for various kinds of partitioning techniques such as range partitioning, list partitioning, hash partitioning, etc. [46]. These methods in general form a mapping between the rows and the partitions that are created. The created partitions are either put on a single machine (single-node partitioning) or on several machines (multi-node partitioning) [4] depending on the application requirements. Multi-node data partitioning is generally considered in order to increase the quality of services in a data grid by locating data near to the location where it is used most. The well-known forms in physically designing partitions, used by most commercial vendors supporting horizontal partitioning are as follows:

- *Range Partitioning:* This partitioning technique is used to assign rows to partitions according to a predefined logical range on values of selected partitioning columns like start/end/transaction dates.
- *Hash Partitioning:* This partitioning technique separates rows using a hash function that is applied on the values of partitioning columns like primary keys.
- *List Partitioning:* This partitioning technique decomposes data matching the values of selected partitioning columns with a predefined list of column values.
- *Round-Robin Partitioning;* This partitioning technique allows data to be evenly distributed over partitions. This partitioning scheme is good for queries scanning relations as a whole whereas; not effective for point or range queries due to the random distribution of data.
- *Composite Partitioning:* Several of the single (one level) partitioning techniques above

can be applied one after another to form a composite partitioning on the table. For example; after creating partitions of a table with range partitioning; hash partitioning can be applied on the current partitions to form further new partitions [8, 43, 4].

From an academical point of view; the literature contains studies for horizontal data partitioning in several contexts: centralized traditional databases (RDBMS or OODBMS), distributed databases, parallel databases and relational data warehouses [4]. Centralized databases consider the problem of horizontal partitioning as selecting both the set of relations that should be decomposed and the most beneficial fragments associated with that selected relations, in an effective way in order to minimize the total querying costs compared to non-partitioned case. Distributed and parallel databases; in addition to the fragmentation process in centralized databases, also considers the allocation and possibly replication of fragments to sites. The design of decomposition, allocation and replication of fragments has important impacts on the query processing [4].

For horizontally decomposition problem, several query-driven approaches have been proposed to develop partitioning techniques that are categorized as: predicate-based, affinity-based and cost-based [47]. Predicate-based algorithms consider the set of predicates used by the most frequently asked queries in forming the partitions of a relation [44, 43]. Affinity-based algorithms try to improve predicate-based algorithms by producing sets of high-affinity predicates based on the total access frequencies of queries. Partitions are formed by the conjunction of the selected predicates [48]. Cost-based algorithms uses a cost model that associates each proposed partitioning scheme with a cost [49]. Other than the well-known decomposition procedures given above some further partitioning studies are; heuristics based [50], knowledge based [51], lattice-structures based [52], partitioning by reference [53] etc.

Another track in the studies of horizontal partitioning arose with the distribution of partitions over multi-nodes. While providing some facilities like shareability, availability, reliability, performance, etc.; distributed database environments also carry with their complexity, cost, security, integrity control, heterogeneity, etc. Data allocation and replication are the next issues after decomposing a non-partitioned relation into fragments in a distributed environment setting. In [54], the replication of data is analyzed in a service-oriented architecture using OGSA-DAI middleware to have a highly available system considering possible failures of some nodes. In [55], two closely related problems, query optimization and data allocation are

explored in a combined fashion to develop a strategy to minimize both of the problem costs.

As a last but not the least important problem of issues studied in the literature for horizontal partitioning in distributed environments is migration and query processing. Migration is the change in the partition assignment of a row by an update on the values of partitioning columns. The problem is important especially for databases with high-availability requirements. The partitioning strategies may effect and possibly trigger the migration problem. For example; since names of people are less prone-to-change compared to the currency balances of people; it is wiser to partition data according to names rather than the currency balances where migration would be inevitable. An example of the migration problem in query-intensive, high-available applications is [56] which presents two policies: one with central controller regulating the migration according to the query loads of other servers and the other with letting individual servers to regulate their data by selecting a partner for migration to balance query loads.

Distributed query processing (DQP) as mentioned in Section 2.1, is rather a broad and an advanced issue that has been explored considerably. DQP has been dealt with several aspects for horizontally partitioned relational databases in a distributed environment. Efficient query processing algorithms for costly operators especially for join, have been developed. In [57], three models (minimal response time, minimal total execution cost and hybrid of two) have been formalized for use in join operations of fragmented relations in order to minimize the elapsed time perceived by the user. In [58]; local and remote semi-join strategies are introduced in order to decrease the communication costs involved when joining the distributed partitions, and a mathematical model for remote semi-joins (i.e. for the semi-joins taking part in an arbitrary site) is developed. In [59], set queries (queries involving set operations) are considered with horizontally partitioned database systems and a mathematical model is developed together with solution strategies. Additionally; there are studies that horizontally partition the data on the fly for query processing independent from the partitioning of the actual data. For example; in [60], according to the functional dependencies, the horizontal partitioning of relations has been considered in the execution of cyclic queries. In [61] join ordering, which is an important factor in queries with multi-joins, has been analyzed with horizontally-hash partitioned relations in order to reduce costs of repartitioning of relations during query processing. Figure 2.4 shows the use of horizontal data partitioning throughout the database evolution [4].

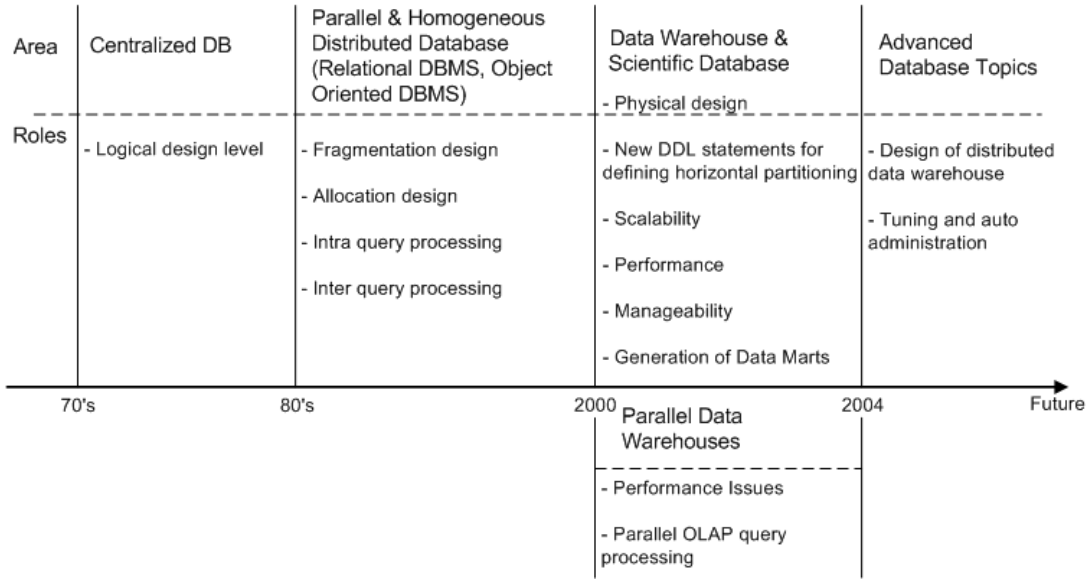

Figure 2.4: The use of horizontal partitioning along the database evolution [4]

RUPAGATION, is a well-defined study on replica management. Although, the target problems and the initial environment of this thesis and system RUPAGATION [62] are completely in different spots; discussing it will clear the separation of this thesis study from such replica consistency models. RUPAGATION deals with keeping distributed replicas of a database consistent throughout the update requests in a service-oriented data grid environment. The system is based on OGSA-DAI middleware and implements both primary replica consistency model (where data updates are applied first on the primary replica and then propagated to the secondary replicas) and two-phase commit protocol (where the updates are applied as a whole in all the replicas if and only if all the replicas are available and ready). The interesting service of RUPAGATION, replica update propagation service (RUP), consists of; replica update manager, update construction, update applier and update transfer modules. The update construction module is responsible for producing update sets for each update request coming from the secondary replica sites. The update applier module is called by the replica update manager and is responsible for performing updates on the requested data resource. The update transfer module is used in transmitting update sets to other replica sites. The replica update manager is responsible for managing the status information about the local replica and notifying the consistency service above the RUP service layer about the received and produced update sets at the local replica. The experimental results of RUPAGATION show that, keeping replica sites consistent in a distributed environment is feasible when updates are collected and

committed at once that is when update sets are big enough to commit. Update sets with single update requests seems to take a considerable amount of time compared to non-replicated case due to the inefficient transmission protocol under, SOAP.

Recent studies [42] provide examples of using DQP to integrate horizontally partitioned data via views, we understand that the integration process described in those examples deals with disjoint partitions only. Besides, the integration is achieved through a union query that constructs the view as a static DQP resource, which inevitably limits the flexibility of the overall data integration environment since those views have to be created as part of the environment setup. Another possible alternative approach to integrate partitioned data is to use *SQLBag* activity of OGSA-DAI, which accesses multiple data resources by posing the same query against them and merges the results. Similar to the views approach *SQLBag* activity solution can deal with disjoint partitions only. That is, to our knowledge, there exist no study on the target problem of this thesis; on horizontal partitions of RDBMS that are replicated and distributed to several shared-nothing nodes once and then emerged to have overlapping data sections with possibly different data updates of different applications without propagation in course of time in a distributed service-oriented setting.

CHAPTER 3

EXTENDING OGSA-DAI DQP TO SUPPORT PARTITIONED DATABASES

3.1 General Considerations on the Extension

In some environments; the data is distributed by replication to different points for easy access from different locations by possibly different applications to decrease the communication overheads and increase the number of clients being served. The applications may operate on different replicas of data. So, any updates or insertions to data may reside in one replica whereas the other replicas remain unchanged. The distributed database management systems (DDBMS) has several strategies to solve this problem generally by maintaining consistency between replicas. For the environments where the installation of a DDBMS is problematic, costly or even almost impossible (e.g. scientific studies working with independent databases on heterogeneous systems); the management of the partitions and accessing the last updated version of data with a single interface to all partitions are issues to be resolved. In grid environments where heterogeneity is an issue to be handled; the management of replicas is more complex and keeping all replicas 100% synchronous simultaneously is not that efficient [63]. When the overhead of replica consistency is considered, propagating update operations among replicas may be discarded in some systems with the requirements of high data replica count and frequent data update requests. So for such systems; although some data will continue to exist exactly the same for some replicas, replicas will no longer stay as replicas. They will be more like as a configuration of multi node partitioned database with possibly partial replicated data. A similar problem is the migration problem of multi-node partitioned databases. As mentioned in Chapter 2.4, migration problems occur when an update operation requires data movement from one partition to another due to changing properties of data according to

the partitioning criteria. Although less likely to occur; multi-node partitioned systems with data updates causing migration problems frequently, may discard to migrate data from one partition to another in order to eliminate the migration overhead involved with the updates or keep the position of data constant. Then the partitions will grow or shrink separately; which will again cause the partitions to have separate or intersecting data records referring to the same real world entity with different field values and update timestamps. We call the state of two partitions (i.e. items that were previously replicas, but emerged to be non-migrated data), which both partitions have tuples with common primary key values representing the same real world entity with possibly different values other than the primary keys, as overlapping. On the other hand, if the partitions have nothing in common, that is there are no tuples with common primary key values (i.e. primary keys group), they are called as disjoint. The problem this thesis targets is retrieving the most up to-date data from multi-node horizontally partitioned relational databases in a service oriented environment enhancing the well-known OGSA compliant middleware in an effective manner. Actually, the problem can be seen as a special case of OGSA-DAI DQP usage in accessing and integrating databases with homogeneous schema. The service-oriented architecture OGSA-DAI DQP; allows access to each horizontal database partition as if they are completely different, separate databases and the client has to be informed of the partitioning. Thus the client has to manage the overlapping data in all these partitions either by writing long queries to extract the lastly updated data or executing the same query for different partitions for several times. This process can be painful when complex queries involving join operations are considered. Such operations may also cause the processing of possibly the same data (due to replicated data in different partitions) several times. Also the benefits of having horizontally partitioned data may not be used in executing queries. Additionally, different from DDBMSs, since the selected environment and the underlying middleware are service-oriented, we do not require the partitions of tables or databases reside on the same DBMS trademark. That is; it does not matter whether the partitions are kept heterogeneously or homogeneously for this thesis.

In this thesis; some assumptions have been made about the properties of the partitioned data in order to be able to concentrate on the real problem and to define the scope of the problem. Some of these assumptions can be removed with small efforts without changing the vision of the study. The assumptions about the environment are as follows:

1. *Uniqueness of Primary Keys:* The primary keys are unique throughout the partitions.

This means; no two different real world entity may be referred by using the same primary key, throughout the partitions. Each world entity has its own primary key value.

2. *Timestamp Knowledge:* All the tables in databases have a constraint on having a *timestamp* column that is updated according to the update times of the rows. The column information (name, position in tuple etc.) of *timestamp* is known.

3. *Identical Schemas:* The names of the columns and tables are identically same in all the partitions. This assumption can be removed easily by producing different identifiers using IP number, database name, etc. like the action for identical table name resolution of OGSA-DAI DQP system in gathering schemas [3].

4. *Partitioning Information:* The overlapping/disjoint properties between partitions are mostly known. If the information is not available; the partitions are accepted as overlapping. To remove the assumption; the incomplete information of disjoint/overlapping can be gathered by executing queries on partitions of a table/database several times and producing related histograms on primary keys of partitions.

Throughout the thesis, partition will refer to a single partition of a horizontally partitioned relational table. If other type of partitioning (vertical, hybrid, etc.) is implied; it will be explicitly stated. As a solution; in order to preserve the main flow of the execution of OGSA-DAI DQP system mentioned in the related work section; a new algebraic operator with two different implementations based on the partitioning information is proposed to handle the update problem between partitions on-the-fly, when data on partitions are accessed. The main concern of the new operator is both allowing clients to ease the integration of the partitioned tables involved in the queries in a seamless manner and eliminate not up to date data as early as possible to decrease unnecessary data movements. The new operators are designed as binary and n-ary in terms of the number of inputs; *UnionPartitions* and *UnionPartitionsNary*. In the query plan; *UnionPartitions* (i.e. both versions, binary and n-ary) are located above the scanning operators for the partitions of the corresponding tables that are involved in the query execution. So, the inputs to *UnionPartitions* (i.e. both versions; binary and n-ary) can be scan, projection, rename or *UnionPartitions* operators depending on the requested query.

3.2 Extending OGSA-DAI DQP

3.2.1 Extending OGSA-DAI DQP Federation Description

Some additional information is needed about tables involving in a query like whether they are partitioned or not and if partitioned with which partition they are overlapping/disjoint. Since MySQL[1] is not capable of supporting this type of information, the mediator of the system (i.e. OGSA-DAI DQP services) is slightly manipulated to load partitioning information from a given XML file. So after populating OGSA-DAI DQP system with the information of the schemas of the resources, the partitioning information of the concerned tables are added to the data dictionary to be used during query processing. An example of a partitioning information file that should be located for the use of OGSA-DAI DQP system is illustrated in Figure 3.1. The tags in the partitioning XML file are organized as follows;

- *partitionInfo:* This tag represents the root element of the document under which partitioned tables will be defined.
- *partitionedTable:* This tag represents a partitioned table. The *name* attribute stands for the title to be used in queries to refer to the partitions as a whole.
- *partition:* Each partition of a partitioned table is represented by this tag. The *name* attribute stands for the title of the partition (i.e. table name for that partition), *resource* attribute stands for the title of the resource name deployed on OGSA-DAI DQP system and finally; *id* attribute stands for the identification of the partition.
- *disjoint:* The concern of complete separateness between two partitions is represented by this tag. The *id* attribute refers to the identification of the partition that is disjoint with the current partition.
- *overlap:* The concern of overlapping between two partitions is represented by this tag. The *id* attribute refers to the identification of the partition that is overlapping with the current partition.

The partitioning information is used in order to determine the ordering and inputs of the newly proposed *UnionPartitions* operator; when the query requires the involvement of a partitioned

[1] MySQL 5.0.15

```
<partitionInfo>
  <partitionedTable name = "Partitioned_users1">
      <partition name="users1" resource="usersP1" id="1">
            <disjoint id="2"/>
            <overlap id="3"/>
    </partition>
    <partition name="users1" resource="usersP2" id="2">
<disjoint id="1"/>
<overlap id="3"/>
    </partition>
    <partition name="users1" resource="usersP3" id="3">
<overlap id="1"/>
<overlap id="2"/>
    </partition>
  </partitionedTable>
  <partitionedTable name = "Partitioned_users2">
        <partition name="users2" resource="usersP1" id="1">
        <disjoint id="2"/>
        <overlap id="3"/>
    </partition>
    <partition name="users2" resource="usersP2" id="2">
<disjoint id="1" />
<overlap id="3" />
    </partition>
    <partition name="users2" resource="usersP3" id="3">
<overlap id="1" />
<overlap id="2" />
    </partition>
  </partitionedTable>
</partitionInfo>
```

Figure 3.1: Example Partition Information

table. The state of partitioning properties of all partitions in an environment setting (e.g. federation, grid) is called as a partitioning configuration. For example; disjointness of all partitions, the overlapping of all partitions, etc.

3.2.2 Extending the Optimization Chain and Activities

As mentioned in section 2.2.3, OGSA-DAI DQP middleware transposes the generated logical query plan through an *Optimization Chain* in order to optimize, schedule and fragment the query plan and thus, to convert it from a centralized plan to a multi-node evaluation plan. The optimization chain contains an ordered sequence of optimisers. In the general sense; an optimizer is a functionality that takes in the query plan from the anterior optimiser and outputs a modified query plan according to its assigned task to the posterior optimiser. There are also items that are not optimizers but reside in the optimization chain in order to perform several tasks during plan processing. For example, *VisualiseOptimiser* does not modify the query plan, instead renders the received query plan to a DOT file.

```
<optimisationChain>
  <optimiser class="uk.org.ogsadai.dqp.lqp.optimiser.QueryNormaliser" />
  <optimiser class="uk.org.ogsadai.dqp.lqp.optimiser.select.SelectPushDownOptimiser" />
  <optimiser class="uk.org.ogsadai.dqp.lqp.optimiser.rename.RenamePullUpOptimiser" />

  <optimiser class="uk.org.ogsadai.dqp.lqp.optimiser.project.ProjectPullUpOptimiser" />
  <optimiser class="uk.org.ogsadai.dqp.lqp.optimiser.join.JoinOrderingOptimiser" />

  <optimiser class="uk.org.ogsadai.dqp.lqp.optimiser.join.JoinAnnotation" />
  <optimiser class="uk.org.ogsadai.dqp.lqp.optimiser.project.groupby.
                                                InsertProjectAfterGroupByOptimiser" />
  <optimiser class="uk.org.ogsadai.dqp.lqp.optimiser.project.pushdown.
                                                ProjectPushDownOptimiser" />
  <optimiser class="uk.org.ogsadai.dqp.lqp.optimiser.project.redundant.
                                                RemoveRedundantProjectOptimiser" />
  <optimiser class="uk.org.ogsadai.dqp.lqp.optimiser.partitioner.PartitioningOptimiser" />
  <optimiser class="uk.org.ogsadai.dqp.lqp.optimiser.implosion.TableScanImplosionOptimiser">
    <property name="selectOnly.resource" value="SomeResource" />
  </optimiser>
  <optimiser class="uk.org.ogsadai.dqp.lqp.optimiser.join.FilteredTableScanOptimiser">
    <property name="bigtable.min.size" value="1000" />
    <property name="table.size.ratio" value="10" />
    <property name="table.to.filter" value="Resource4_faculty" />
  </optimiser>
</optimisationChain>
```

Figure 3.2: Default Optimization Chain of OGSA-DAI DQP

Like activities, the *optimization chain* is one of the extendibility points of OGSA-DAI DQP middleware. The order, type and implementation of the optimisers can be changed, optimisers can be removed and new optimisers can be added as far as the modified chain produces reasonable query plans. Figure 3.2 shows the default optimization chain defined in OGSA-DAI DQP.

In order to insert the *UnionPartitions* operator to the required locations in the query plan, a new optimiser *HorizontalPartitionsOptimiser* is produced and located to be applied between *ProjectPullUpOptimiser* and *JoinOrderingOptimiser* in the optimizer chain. The default optimization chain is seen in Figure 3.2. The location for the new optimizer is selected in such a way that; the renaming, selection and projection operators are ordered and located beforehand and the optimization of the join-ordering is not issued yet, in case it is effected from the output size of the *UnionPartitions* operator.

Via the *UnionPartitions* operator, the partitioning information of a table is hidden from the user; that is, the client sees all the partitions of a table as a single table and therefore it queries the partitions of a table as a whole. In the execution of the optimizer chain, after the execution of *ProjectPullUpOptimizer*; the query plan contains a single scan operator representing all the partitions of the same table. After taking this query plan; *HorizontalPartitionsOptimiser*,

searches for the scan operators accessing horizontally partitioned tables. Upon finding them; a separate scan operator is created for each of the partitions of that table. At this point; if the *UnionPartitions* operator is selected to be binary; the problem of ordering of these scan operators arises. If the *UnionPartitions* operator is selected to be n-ary, input operators are streamed separately without any concern of operator ordering. Both conditions will be detailed in the following chapter.

Each operator in the query plan has an associated builder class responsible for building activities for that operator. The builder classes are applied in the order of the given query plan. The query plans partitioned from the centralized query plan are submitted to builder classes for the analyses and production of activities. The builder classes, take in a query plan and produce an activity pipeline of the operators in that query plan. After the production of activity pipelines for each of the query plan partition, they are executed either locally or remotely according to their previously assigned scheduling properties. For the execution of *UnionPartitions* and *UnionPartitionsNary* operators, two builder classes *HorizontalPartitionsUnionBuilder* and *HorizontalPartitionsUnionNaryBuilder* are created and associated with those newly created operators for production and configuration of the related activities, *TupleUnionPartition* and *TupleUnionPartitionNary*. These activities, are the main activities behind the execution of *UnionPartitions* and *UnionPartitionsNary* operators, which will be detailed in the next chapter.

3.3 New Operator Design

From a conceptual point of view, the *UnionPartitions* operator can be defined as a relational operator that takes at least two partitions of a relational table and outputs a relation containing up-to-date tuples which are identified based on the primary key values. Although there is no restriction on the algebraic operators accepted by the *UnionPartitions* operator as input; they are generally select, project, rename or *UnionPartitions* operator.

Unlike the basic content-neutral relational operators, the new *UnionPartitions* operator is content-specific in that it requires the timestamp of a tuple; since it is designed to manage partitions or replicas of tables (possibly containing overlapping data) based on their update times. It is primitive in that it performs one single operation (i.e. discriminating against

timestamps and primary key values) and leaves other primitive operations like projection (i.e. timestamp column projection) to other formerly implemented primitive operators. It provides a virtual view of disjoint/overlapping partitions of a relation, allows clients to write queries over this view and guarantees the result to contain safe data in terms of timeliness by making comparisons among partitions. The *UnionPartitions* operator is implemented and used in three forms:

- *Binary:* Takes in tuples coming from two input operators and outputs tuples like any other binary operator. During the selection of up-to-date tuples; BKDR hashing function is enhanced for tuples coming from left and right tuples. Right tuples wait for the completion of hashing of left tuples. The tuple matching and output production starts only after the receipt of all tuples from the left input.

- *Strict N-ary:* Takes in tuples from n inputs simultaneously and locates the received tuples into separate hash tables while at the same time checks for the timeliness of tuples.

- *Hybrid:* Actually, it is not a separate operator implementation, it is the behaviour of the system using binary and n-ary operators together in the same tree. As mentioned, the optimiser *HorizontalPartitionsOptimiser* is responsible for updating the query plan by inserting *UnionPartitions* operator to the required locations according to the binary, n-ary or hybrid behaviour choice defined in a separate file. If this hybrid behaviour is selected; then internally overlapping, externally disjoint sets are constructed using the partitions of the table. Then elements of each set is combined to form binary trees of the *UnionPartititons* operator. Finally, the trees from each set are combined to form an n-ary *UnionPartitions* tree. Since the inputs are disjoint, the output production is possible as soon as the tuples are received from the inputs.

The details of these three forms are given in Chapter 4.

CHAPTER 4

THE DESIGN AND IMPLEMENTATION OF THE *UNIONPARTITIONS* OPERATOR

4.1 *UnionPartitions* as a Binary Operator

Since the ordering of some of the operators (e.g. join) in a binary query plan tree may effect the response time of a query dramatically, query optimization techniques have to arrange the inputs of the operators in an efficient manner. There is the need to arrange the inputs of the *UnionPartitions* operator, while changing the intermediate query plan produced by the OGSA-DAI DQP system to deal with the horizontally partitioned tables. This need arises from two reasons:

- The partitions of a table have relational properties of being overlapping or disjoint; therefore the early involvement of overlapping partitions in a *UnionPartitions* process may decrease the amount of data carried to upper operators.
- The environment that the operator being used is distributed where it is important to minimize data transmissions between operators.

The literature does not contain any striking study on the ordering of the binary operators *UnionAll* and *Union*, which are somewhat resembling the *UnionPartitions* operator. The related work primarily focuses on the ordering of the *Join* operator, since it is the most time-consuming, widely used binary operator by database applications. The operator *UnionPartitions*, that is introduced in this thesis, is a new operator; therefore there is not any ordering algorithm yet. In order to develop an ordering algorithm for the *UnionPartitions* (binary case) operator, several studies on the ordering of *Join* operator have been studied [64, 65, 66, 67].

The ordering problem for the *UnionPartitions* operator, may resemble to the join ordering problem in some respect, if the pre-mentioned problem dimensions and assumptions are considered. This section explains the details of the binary *UnionPartitions* operator.

4.1.1 Locating the Binary *UnionPartitions* Operator in Query Plan Tree

In the join-ordering problem, the search space increases rapidly as the number of relations in the join increases; because of the commutative and associative property of join operator [64]. The general problem of join-ordering on the number of relations has NP-Hard [64, 66, 67, 68] problem characteristics; therefore, query optimizers try to find not the most optimal but the possible best order by reducing search spaces using some heuristics and limitations on the join tree structures. Similarly, *UnionPartitions* is an associative and commutative operator in which the number of possible processing trees increases by the number of partitions for the table participating in the query. OGSA-DAI DQP system provides the decomposition of the query plan for execution in different evaluators enhancing the pipelined and independent parallelism during the processing of the operators. As a result, the search space of the *UnionPartitions* tree is limited to a bushy-like tree which is well-known for its opportunity of independent parallelism [67].

A deterministic algorithm considering the following rules is developed to form a corresponding bushy-like tree structure for a given partition configuration in a bottom-up fashion;

- Give precedence to overlapping partitions in selecting inputs for the *UnionPartitions* operator in order to eliminate outdated data formerly.
- Push *UnionPartitions* operators with inputs of overlapping partitions downwards and the ones with inputs of disjoint partitions upwards in the partitioning operator tree, as possible as it is, in order to reduce redundant data transmissions.
- Give precedence to the formation of sub-trees with overlapping properties over the formation of sub-trees with lower heights.
- Select the sub-tree with the lowest height as the left child of the *UnionPartitions* operator, in order to execute the *UnionPartitions* operator as soon as possible.

The concept of the *operator tree segment* is introduced in the ordering of *UnionPartitions*

operators. An *operator tree segment*, shortly called a *cell* for convenience, is defined in terms of an operator tree, a list of partition ids used in that operator tree, a list of partition ids that are overlapping with the partitions in the list of used partitions and the height of the operator tree in this cell. A list of cells is called a partition matrix. Figure 4.1 gives the pseudocode of the algorithm for the ordering of binary *UnionPartitions* operators.

First of all, a list of initial cells(i.e. the initial partition matrix), including a cell for each partition of the partitioned table is created. In an initial cell;

- *Operator Tree*; is either a scan operator over a partition or a project operator or rename operator with input of scan/project/rename operator. This operator tree is formed according to the unary operators (that should be pushed down) above the scan operator of partitioned table in the query plan received by the *HorizontalPartitionsOptimiser*.

- *Used Partitions*; contains only the *id* of the current partition.

- *Overlapping Partitions*; contains the partitions that are overlapping with the current partition.

- *Height* of cell's operator tree; is given to be 0.

According to the query requested for execution; there may be project operators just before the scan operators in the initial query plan received by the *HorizontalPartitionsOptimiser*. Since projection is a way of reducing the amount of data carried to upper operators; project operators are generally pushed down on the query plan tree. If such a case is under consideration; during the initial cell list construction; an appropriate project operator is created in addition to the creation of the scan operator on a partition in order to decrease the amount of data flowing up. Since, the *UnionPartitions* operator needs the primary key values of tuples during the elimination of outdated data, the optimizer attaches the primary keys as projected parameters for the newly created projection operators. This situation also holds for the rename operators encountered. The elimination process is explained in Section 4.1.2

```
(1)   Operator buildBinaryUpTree (Operator scanWholePartitionedTable) {
(2)        partitionMatrix  = createInitialPartitionMatrix(scanWholePartitionedTable);
(3)        while (partitionMatrix.size > 1) {
(4)             findBestFellows(partitionMatrix) -> < C_1, C_2 >
(5)             // Remove C_1, C_2 from Partition Matrix
(6)             // Combine cells C_1, C_2 to C_new
(7)             // Add C_new to partitionMatrix
(8)        }
(9)   }
```

```
(1)   < C_1, C_2 > findBestFellows(partitionMatrix) {
(2)        int minHeight, overlapListCount, cardinality = -1
(3)        int minDisHeight, disCardinality  = -1
(4)        Cell cell, cellFellow, disCell, disCellFellow = null
(5)        for (i = 0 -> partitionMatrix.size) {
(6)             // Find best fellow cell (C_fellow) for cell i (C_i) in partitionMatrix
(7)             findBestFellowCellToUp(partitionMatrix, C_i) -> C_fellow;
(8)             // Calculate height, overlapListCount and cardinality of the cell
(9)             // produced by combining <C_i, C_fellow>, as temporary values; tmpHeight,
(10)            // tmpOvListCount and tmpCardinality
(11)            if (C_i DISJOINT C_fellow) {
(12)                 if (minDisHeight > tmpHeight || (tmpHeight == minDisHeight &&
(13)                        disCardinality > tmpCardinality) || minDisHeight == -1) {
(14)                      // Assign C_i -> disCell, C_fellow -> disCellFellow
(15)                      // Assign tmpHeight -> minDisHeight, tmpCardinality -> disCardinality
(16)                 }
(17)            } else if (minHeight > tmpHeight || (tmpHeight == minHeight &&
(18)                        ((overlapListCount < tmpOvListCount) ||
(19)                     (overlapListCount == tmpOvListCount && cardinality > tmpCardinality)) ||
(20)                      minHeight == -1) {
(21)                         // C_i -> cell, C_fellow -> cellFellow
(22)                         // Assign tmpHeight -> minHeight, tmpOvListCount -> overlapListCount
(23)                         // Assign tmpCardinality -> cardinality
(24)            }
(25)       }
(26)       return <cell, cellFellow>;
(27)  }
```

Figure 4.1: Pseudocode for Ordering Binary UnionPartitions Operator

After the initial partition matrix is prepared, it is searched to find two cells, C_1 and C_2, which will combine best based on the rules given above. During the selection of C_1 and C_2, first the binary combinations of overlapping cells are checked. If it is not available, then disjoint cells are checked. Two cells with the following properties are considered respectively;

- The lowest combined tree height, in order to preserve the bottom-up approach and bushiness.
- The highest combined overlapping list, in order to minimize both the transmission cost and timestamp matching operation costs of overlapping data.
- The lowest combined relation sizes, in order to minimize the transmission cost of tuples that will be carried among middle-layer *UnionPartitions* operators up to the root.

Upon finding, C_1 and C_2, a new cell is formed with the following fields:

- *Operator tree:* The root of the operator tree is the *UnionPartitions* operator whose inputs are previously selected cells; C_1 and C_2. The cell with the lowest cardinality is selected to be the left input in order to output result tuples as soon as possible during the execution of the *UnionPartitions* operator which is detailed in Section 4.1.2.
- *Used partitions:* Set-union of the used partitions lists of C_1 and C_2.
- *Overlapping list:* Set-union of the overlapping lists of C_1 and C_2.
- *Height:* maxHeight (C_1, C_2) + 1

Next, the newly created cell is inserted to the list of cells(i.e. the partition matrix) and previous cells, C_1 and C_2, are removed. This cell combination process continues until there is a single cell, with the operator tree root *UnionPartitions* is left in the partition matrix. Then, a project operator is created to remove the unrequested fields from the resulting tuples (i.e. possibly the timestamp or primary key fields) and located above the root *UnionPartitions* operator. Finally, the resulting operator tree with the root project operator takes the place of project/scan/rename operator over the corresponding partitioned table in the query plan.

Consider the following example on the ordering of the *UnionPartitions* operator, for a simple scan query on a table with 5 partitions. Assume that the following relationships (i.e.

overlapping or disjoint) between partitions occur (where unmentioned relationships between partitions means they are disjoint);

Table 4.1: Example Partition Information

Partition Identifier	List of Overlapping Partitions	Cardinality (x1000 tuples)
P1	P3 , P4	30
P2	P5	30
P3	P1 , P4	60
P4	P1 , P3	40
P5	P2	25

The algorithm performs the following steps to produce a *UnionPartitions* operator tree:

1. Initially, the partition matrix in Table 4.2 will be produced. A row of the partition matrix is called a *cell*. The cells contain information about sub-trees that will be considered as inputs during the creation of *UnionPartitions* operators. The list of used partitions will be enhanced to refer to cells. For instance, the cell with the used partitions list of P1, P3 will be referred as P1P3.

Table 4.2: Initial Partition Matrix

Operator Tree	List of Used Partitions	List of Overlapping Partitions	Operator Tree Height	Cardinality
π (σ P1)	P1	P3 , P4	0	30
π (σ P2)	P2	P5	0	30
π (σ P3)	P3	P1 , P4	0	60
π (σ P4)	P4	P1 , P3	0	40
π (σ P5)	P5	P2	0	25

2. Next, a best fellow cell is seeked for each of the cell in the partition matrix. The fellow cells are firstly seeked from the list of overlapping partitions. If the cell is to be a disjoint one or the overlapping partitions are already used up then other disjoint partitions are considered to find a fellow cell. For example, the cell P1 has two overlapping partitions; P3 and P4. The algorithm will consider both partitions as the best candidate fellow of P1 and select the one with the lowest tree height, the highest overlapping list size and the lowest cardinality. Tree heights (0) and overlapping list sizes (2) of cells P3 and P4

are equal. So, P4 will be selected as the best fellow for cell P1 for its low cardinality, and we will represent this fellowship as P1-P4. Similarly; the following fellowships will be produced; P2-P5, P3-P1, P4-P1 and P5-P2.

3. Then, the best fellowship is selected from the candidate fellowships according to the lowest combined tree height, the highest total of overlapping lists and the lowest total of cardinalities.

   - P1-P4; Height 1; Overlap lists total size 4; Total record size 70
   - P2-P5; Height 1; Overlap lists total size 2; Total record size 55
   - P3-P1; Height 1; Overlap lists total size 4; Total record size 90
   - P4-P1; Height 1; Overlap lists total size 4; Total record size 70
   - P5-P2; Height 1; Overlap lists total size 2; Total record size 55

4. As a result, P1-P4 fellowship is selected for its lowest height. The cells P1 and P4 are removed from the partition matrix and a new cell is produced. Since *cardinality(P1)*< *cardinality(P2)*, P1 is selected as the left child of the *UnionPartitions* operator which is then assigned to be the operator tree of the newly created cell. The new cell is added to the partition matrix to form Table 4.3.

Table 4.3: Updated Partition Matrix - 1

Operator Tree	List of Used Partitions	List of Overlapping Partitions	Operator Tree Height	Cardinality
π (σ P2)	P2	P5	0	30
π (σ P3)	P3	P1 , P4	0	60
π (σ P5)	P5	P2	0	25
UP(π(σ P1)) (π (σ P4))	P1, P4	P3	1	70

5. The algorithm then iterates over the steps 1-4 until there is a single cell left in the partition matrix. The rest of the iterations are given below;
   Iteration:
   Candidate Fellowships:

   - P2-P5; Height 1; Overlap lists total size 2; Total record size 55
   - P3-P1P4; Height 2; Overlap lists total size 3; Total record size 130

- P5-P2; Height 1; Overlap lists total size 2; Total record size 55
- P1P4-P3; Height 2; Overlap lists total size 3; Total record size 130

The fellowship of P2-P5 is selected, cells P2 and P5 are removed and a new cell is added to the partition matrix.

Table 4.4: Updated Partition Matrix - 2

Operator Tree	List of Used Partitions	List of Overlapping Partitions	Operator Tree Height	Cardinality
π (σ P3)	P3	P1 , P4	0	60
UP(π(σ P1)) (π (σ P4))	P1, P4	P3	1	70
UP(π(σ P5)) (π (σ P2))	P2, P5	-	1	55

Iteration:

Candidate Fellowships:

- P3-P1P4; Height 2; Overlap lists total size 3; Total record size 130
- P1P4-P3; Height 2; Overlap lists total size 3; Total record size 130
- P5P2-P3; Height 2; Overlap lists total size 2; Total record size 115

The fellowship of P3-P1P4 is selected, cells P3 and P1P4 are removed and a new cell is added to the partition matrix.

Table 4.5: Updated Partition Matrix - 3

Operator Tree	List of Used Partitions	List of Overlapping Partitions	Operator Tree Height	Cardinality
UP (π(σ P5)) (π (σ P2))	P2, P5	-	1	55
UP (π (σ P3)) (UP(π(σ P1)) (π (σ P4)))	P1, P3, P4	-	2	130

Iteration:

Candidate Fellowships:

- P5P2-P1P3P4; Height 3; Overlap lists total size 0; Total record size 185

- P1P3P4-P5P2; Height 3; Overlap lists total size 0; Total record size 185

The fellowship of P5P2-P1P3P4 is selected, cells P5P2 and P1P3P4 are removed and a new cell is added to the partition matrix;

Table 4.6: Updated Partition Matrix - 4

Operator Tree	List of Used Partitions	List of Overlapping Partitions	Operator Tree Height	Cardinality
UP (UP (π(σ P5)) (π (σ P2))) (UP (π (σ P3)) (UP(π(σ P1)) (π (σ P4))))	P1, P2, P3, P4, P5	-	3	185

So for this sample partition configuration, the ordered *UnionPartitions* operator tree is the operator tree of the single cell left in the partition matrix. Figure 4.2 visualizes the operator tree.

4.1.2 The Execution of the Binary *UnionPartitions* Operator

The algorithm for the execution of the *UnionPartitions* operator is rather similar to the *Union* operator with changes at points where *UnionPartitions* handles the update properties of tuples. The implementation for the *Union* operator mainly uses hashing for implementing different union types (*Union*, *Intersect*, *Except*, *Union All* [No hashing used]).

The algorithm for the execution of the *UnionPartitions* operator is encapsulated inside an activity *(TupleUnionPartition)* like any other operator. During processing, *TupleUnionPartition* first looks whether the inputs to *UnionPartitions* are disjoint or overlapping from the type of *UnionPartitions*. If they are disjoint, the two inputs are consumed in separate threads simultaneusly and passed to the parent operator. Since inputs are used up in an interleaving manner, the output results are not ordered according to any field or input.

If the partition type is overlapping, *TupleUnionPartition* (activity for *UnionPartitions* operator) takes its left-hand side tuples, creates a hash table using primary key values and then uses the right-hand side tuples for matching. The matching process can start only after the

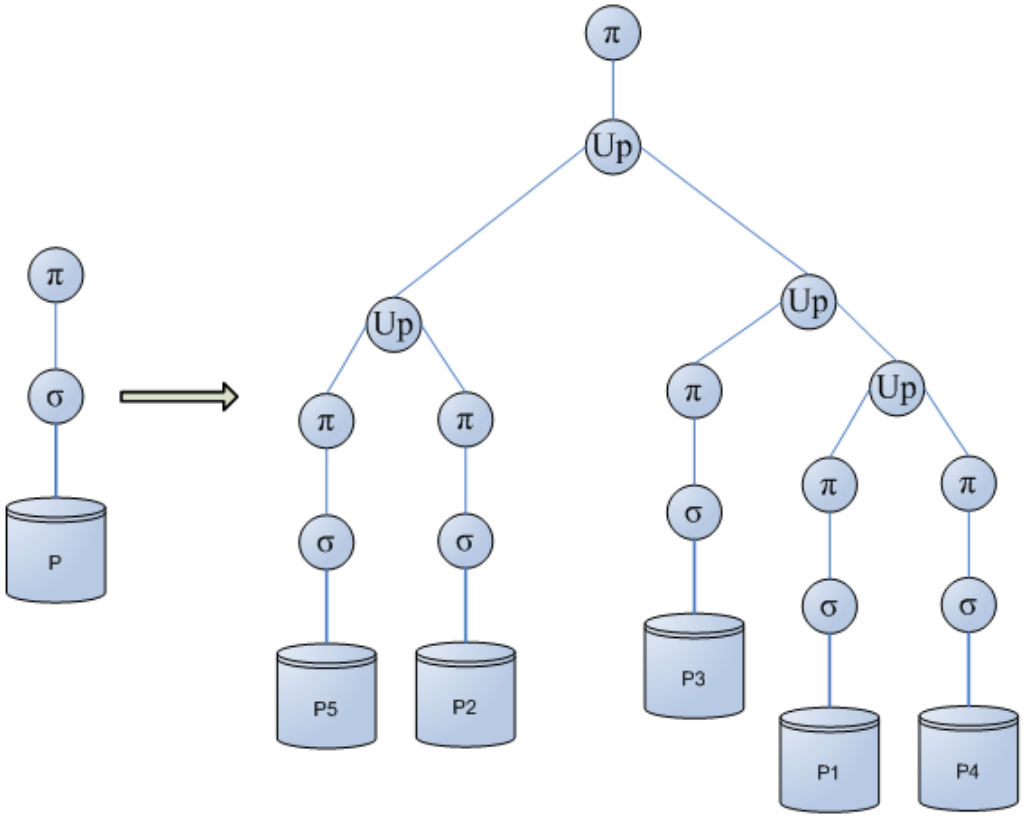

Figure 4.2: An example to UnionPartitions Operator Tree

completion of hashing of the left input tuples. The primary key values of the tuples are used in hashing as a whole (i.e. if exist, values of composite keys are concatenated). During the matching process, when two tuples from two inputs representing the same entity is found, the timestamp of the tuples are checked to get the most recently updated record for that entity. According to that result of timestamp check, the older one is removed from the hashed list and the newer one is passed to the output stream. Since no two tuples with exactly same primary-key values can be received from the same input stream (among left stream or among right stream), the newer tuple can be output without waiting for a match from the other or the current stream. This is an advantage for the binary case of the *UnionPartitions* operator, since by outputing results earlier pipelined partitioning is enhanced. The disadvantage of the binary *UnionPartitions* operator is that, as the number of partitions increases, the path of tuples from leaf node (Scanning Operator) to the root node (root UnionPartitions Operator) also increases. As a result; there may occur redundant data transmissions, especially when all partitions are

disjoint.

The hash function used is the BKDR hash function which is defined in [69]. It uses a set of seeds forming a pattern like 31...31...31 in distributing the data evenly. The reason for using a hash-based algorithm is the opportunity of O(1) + O(n) access time, where O(n) is for going over tuples matching the same hash key. Actually it is possible to reduce the O(n), linear probing cost, by using a second different hash function (which may introduce new hashing costs) or using a couple of search algorithms when the data is sorted (not feasible when data is not sorted in a distributed environment). The projection of tuples which is, the removal of fields that are unrequested by the query-initiator, but carried up to the *UnionPartitions* root operator due to their roles in execution (e.g. unrequested timestamp field) and in identification (e.g. unrequested primary keys) of tuples, is realized on tuples by the projection operator located above the root *UnionPartitions* operator. The pseudocode for the execution algorithm is given in Figure 4.3.

```
(1)   execute () {
(2)        Tuple left, right;
(3)        HashTable hashTable;
(4)        while ((left = receiveLeft()) != null) {
(5)             // Add left tuple to hashTable
(6)        }
(7)        while ((right = receiveRight()) != null) {
(8)             // Get matchList tuple right from hashTable
(9)             for (i = 0 - > matchList.size) {
(10)                 if (matchList[i].PKValues == right.PKValues) {
(11)                      if (right.timestamp > matchList[0].timestamp) {
(12)                           // Output right tuple
(13)                      } else {
(14)                           // Output left tuple
(15)                      }
(16)                      // Remove left tuple from matchList
(17)                 }
(18)            }
(19)       }
(20)  }
```

Figure 4.3: Pseudocode for Execution of Binary UnionPartitions Operator

4.2 *UnionPartitions* as an N-ary Operator

In addition to the binary version of the *UnionPartitions* operator, an n-ary version of the *UnionPartitions* operator called *UnionPartitionsNary* is also implemented. *HorizontalPartitionsOptimizer* is configured to form binary, strict n-ary or hybrid *UnionPartitions* trees which are mentioned in Chapter 3.

4.2.1 Locating N-ary *UnionPartitions* Operator Strictly in the Query Plan Tree

In order to form a strict n-ary tree, the *HorizontalPartitionsOptimiser* creates a single *UnionPartitionsNary* operator for the corresponding partitioned table by using the partitions of the table as inputs. Initially, a list containing an operator tree for each partition is created. The operator trees (i.e. generally scan, project or rename operator as the root) in this list are assigned as inputs to *UnionPartitionsNary* in the given order. The information of disjoint/overlapping relationships between inputs are given to the *UnionPartitionsNary* operator separately to be used in execution, during the elimination of outdated data. Since the inputs to the *UnionPartitionsNary* operator do not contain any *UnionPartitions* or any other binary operator, the approach is called as *strict*. For the example partition configuration given in Section 4.1.1, the *HorizontalPartitionsOptimiser* will produce the operator tree in Figure 4.4 for n-ary approach.

4.2.2 Locating the N-ary and Binary *UnionPartitions* Operators in the Same Query Plan Tree

The hybrid approach forms operator trees for partitioned tables using both *UnionPartitions* and *UnionPartitionsNary* operators. The *UnionPartitionsNary* operator on its own may require a considerable amount of memory especially when large tables are considered, because the elimination process of outdated data is carried out on one single node. In order to form a hybrid tree, the *HorizontalPartitionsOptimiser*, separates the partitions of a table into internally overlapping, externally disjoint sub-sets. The partitions are separated in such a way that; partitions belonging to a subset are overlapping with at least one other partition inside the same sub-set and any two partitions belonging to different subsets are disjoint.
After forming these subsets, the partitions in each subset are processed to build up a binary

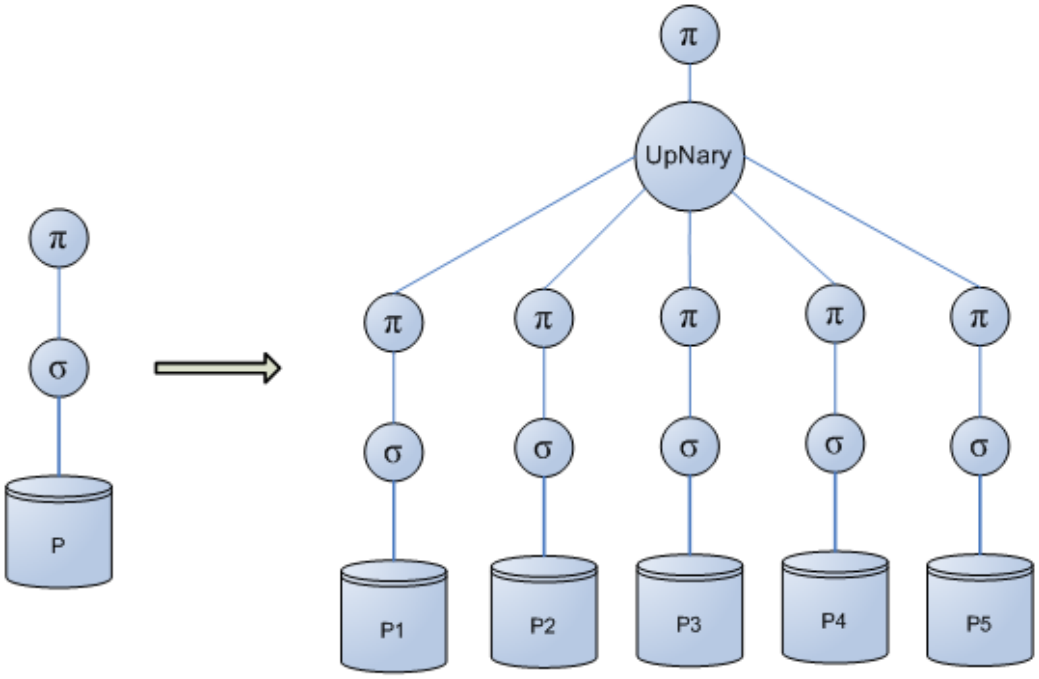

Figure 4.4: An Example to UnionPartitionsNary Operator Tree

UnionPartitions operator tree. The algorithm used to form these sub-trees is exactly same as the one explained in Section 4.1.1.

Then, these externally disjoint sub-trees are given to the *UnionPartitionsNary* operator as disjoint inputs. Since the input streams of the *UnionPartitionsNary* operator are now all disjoint, the execution of the *UnionPartitionsNary* operator will not contain hashing and matching processes. This approach sometimes will produce same output trees with the binary case. For the example given in Section 4.1.1, the hybrid approach would also produce the same output with a different root operator. In this case since P2-P5 and P1-P3-P4 are externally disjoint, the root operator would be *UnionPartitionsNary* operator. Additionally, if there were two more partitions that are disjoint to all others, say P6 and P7, although the binary approach would try to find a good place for them on the binary *UnionPartitions* tree, the hybrid approach would simply locate them as the third and fourth inputs to root *UnionPartitionsNary* as in Figure 4.5.

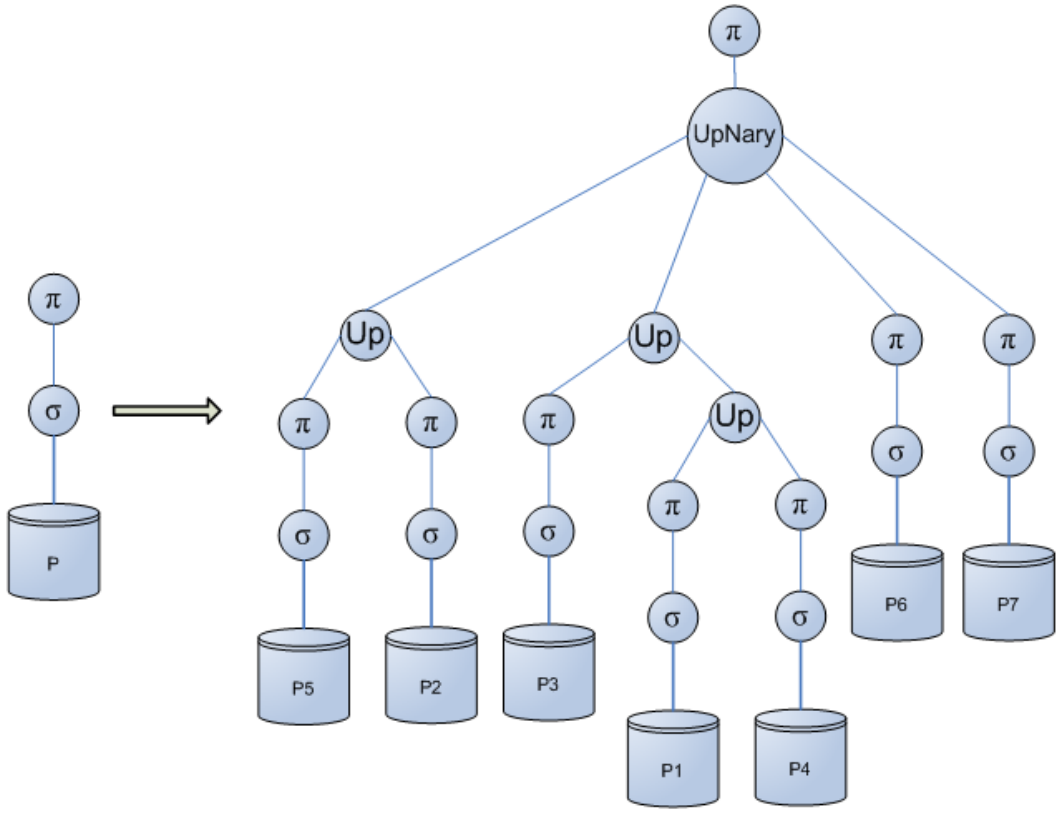

Figure 4.5: An Example to UnionPartitions Operator Tree with Hybrid Approach

4.2.3 The Execution of the N-ary *UnionPartitions* Operator

The execution algorithm of the *UnionPartitionsNary* operator is encapsulated inside an activity (TupleUnionPartitionNary) like any other operator in OGSA-DAI DQP. The functionality of *UnionPartitionsNary* is same as the functionality of the binary case, *UnionPartitions*; eliminating outdated tuples from incoming data streams. The difference comes from the number of inputs a *UnionPartitionsNary* operator handles.

In the n-ary version, data from n inputs are gathered in a threaded way. For each of the inputs a separate thread, one of two types; *Reader* or *Transmitter* is created. The type of a thread for a corresponding input is related to the overlapping/disjoint characteristics of the data tuples received from that input. *Reader* threads are created for inputs of overlapping data, whereas *Transmitter* threads are created for inputs which are disjoint to all other inputs.

Reader threads pull data upon receipt and then send it to the main thread either for locating in a hash table or discarding from the list of output tuples, according to the timestamp and primary

key values of the tuples. Note that, the primary keys are unique throughout the channel, due to the basic rule of primary keys uniqueness throughout a single table in a database. As a result, there is a separate hash table for each input stream carrying in overlapping data, in order main thread not to process a tuple against tuples received from the same input channel and waste time looking for a match in keys. As in the binary case, BKDR hash function is used in the formation and querying of these hash tables.

Each *Transmitter* thread as the name implies, pulls data upon receipt from inputs and sends data to the parent operator as the output of the *UnionPartitionsNary* operator. In this way, the output data can be generated and sent to upper levels without waiting for the full completion of the execution in the *UnionPartitionsNary* operator. Since the input data is processed by the help of threads and hash tables, there is no order of data on the output.

Actually, two different thread sets, *Reader* (of overlapping inputs) and *Transmitter* (of disjoint) sets are formed. Upon completion of these two thread sets, the hash tables of the overlapping inputs containing up-to-date tuples are sent as output to the parent operator of the *UnionPartitionsNary* operator by projecting tuples to remove unrequested fields (i.e. possibly columns of timestamp and primary keys) similar to the explained process in the binary case, *UnionPartitions*. The idea of forming separate hash tables for each input stream is inspired from the MJoin algorithm in [70]. However the application order of overlapping hash tables is not considered, since different from join, *UnionPartitions* have to keep a tuple until it is processed by all the overlapping inputs.

CHAPTER 5

EXPERIMENTS and EVALUATIONS

5.1 Data Preparation

For the purposes of this thesis, we needed chunks of controlled data that are organized under tables to be partitioned and distributed over different databases located at different sites. In order to be able to control tuple data (in terms of the number of tuples or number, type and size of fields etc.); we used *DBMonster* [71] to produce the required data in a random manner. *DBMonster* is a tool that is developed especially to test the performance of database applications with variable database loads. It also helps in adjusting the database structure and index usage for better functioning. The file given in Appendix A is used during the generation of data. Three relations (tables) are produced with rather simple schemas which represent a company database to indicate the allocation of employees to projects. The schemas are shown in Tables 5.1, 5.2 and 5.3. In the tables, column types are given with the types in MySQL [1] and primary keys are italized.

After the generation of data, tuples in tables are written to heaps of files (corresponding to each partition) in the form of SQL statements and databases are created in sites based on the requirements of test cases that will be mentioned in Section 5.3. Finally, each database corresponding to a partition in sites (i.e. a site may contain more than one database) is loaded with the generated data using the prepared files and deployed as a DQP resource on to OGSA-DAI DQP framework.

[1] MySQL 5.0.15

Table 5.1: Schema for Users Tables

	Column Name	Column Type
	userID	bigint(20)
	userName	varchar(8)
	email	varchar(50)
	timestamp	timestamp
Average Row Size (K)	93.03	

Table 5.2: Schema for Projects Tables

	Column Name	Column Type
	projectID	bigint(20)
	description	varchar(255)
	timestamp	timestamp
Average Row Size (K)	190.2	

5.2 Test Environment Setup

During the implementation and evaluation phases of this thesis, we made use of the following resources and software tools.

- Computational Resource: In order to have a realistic (or somewhat close to real) distributed environment, we facilitate five shared-nothing computers nicknames from T1 to T5 (i.e. to be used throughout this thesis). All the nodes are Intel(R) Xeon(R) CPI X7350 64 bits, each with a 2.93 GHz CPU, 32 GB of RAM and link speed of 1Gbps. All data sources are hosted at MySQL databases.
- Auxilary Software Tools: The versions of the following tools are selected according to

Table 5.3: Schema for Allocations Tables

	Column Name	Column Type
	allocationID	bigint(20)
	userID	bigint(20)
	projectID	bigint(20)
	toDate	timestamp
	timestamp	timestamp
Average Row Size (K)	77.0	

the compatibility with the OGSA-DAI DQP[2] framework.

- Globus Toolkit[3]: Globus Toolkit is an open-source technology for Grid environments with software facilities of security, information infrastructure, resource management, data management, communication, fault detection and portability. Globus Toolkit is assembled of components that can be used independently or jointly in developing applications [72]. Actually; the selection of the use of Globus Toolkit depends on future expectations and studies that can be related to more management on resources in a grid environment. For this thesis, the WSRF-Compliant WS Java Container component of Globus Toolkit, Java WSCore, is enhanced as a container to deploy OGSA-DAI DQP framework and resources. Java WSCore contains an implementation of Web Services Resource Framework (WSRF) and Web Service Notification (WSN) specifications. Java WSCore is installed on all the five nodes. Although this thesis is developed and tested using Globus Toolkit, it does not use Globus Toolkit specific functions: Therefore the study can easily be deployed and used with direct Tomcat installations.
- Tomcat[4]: Apache Tomcat is an open-source implementation for Java Servlet and JavaServer Pages [73]. This thesis uses Tomcat for deployment of the Globus Toolkit component, Java WSCore. Since Tomcat is a widely-used web container for Java-based web-applications, OGSA-DAI DQP system also uses it and in turn we used it to develop, deploy and use the extension. Tomcat container is installed on all the five nodes.
- MySQL[5]: MySQL is an open-source relational database management system running as a server providing multiple storage engines (e.g. MyISAM, InnoDB), grouping commit and supporting multi-users [74]. MySQL is used for the management of databases required for this thesis. MySQL is selected for its ease of use, non-commercial support and settled development. But it is possible to use different database management systems if appropriate drivers for JDBC connections exist. MySQL is installed on all the five nodes. According to the requirements of some test cases, a second MySQL DBMS is installed in addition to the first one.

[2] OGSA-DAI DQP 3.2.2
[3] Globus Toolkit 4.0.8
[4] Apache Tomcat 5.5.26
[5] MySQL 5.0.15

– Software Client: In order to execute the queries for several times automatically, an OGSA-DAI DQP software client, DQP Test Manager, is developed. A main view for DQP Test Manager is given in Figure 5.1. DQP Test Manager allows users to:

  * Access and view pre-deployed DQP resources
  * Execute queries one by one manually
  * Execute a file of queries with execution properties defined in an XML file
  * Visualize graphics and execution results related to the execution times of queries

  In order to acquire several measurements on OGSA-DAI DQP extended framework and the newly introduced operator implementations, we required a software to perform the executions one after another, collect the execution times and visualise the results in a systematic way. In order to achieve this, DQP Test Manager configured to accept query files in which total number of executions, number of warm-up executions, graphics to be drawn etc. are defined using an XML schema. Details for configuration of query execution test files and several other screenshots are given in Appendix B.

Four of the computational nodes (T1, T2, T4 and T5) are used as database servers and OGSA-DAI DQP services are located wrapping these databases. The OGSA-DAI DQP services in these nodes are also used as evaluation services. The fifth node (T3) contains an OGSA-DAI DQP service with a deployment of the DQP resource to where the queries are posed to and a client to start the querying process. Several batch files are used in loading databases and deploying OGSA-DAI DQP services and resources on to Globus Toolkit and Tomcat containers.

5.3 Experiment Design

The performance of the extension is measured in a number of ways: *variance in data size of inputs, overlapping data proportion of partitions, number of partitions, number of overlapping partitions and extension scalability*. Each of these five issues is performed as a test case (experiment) by changing data values and partition configuration in a controlled way.

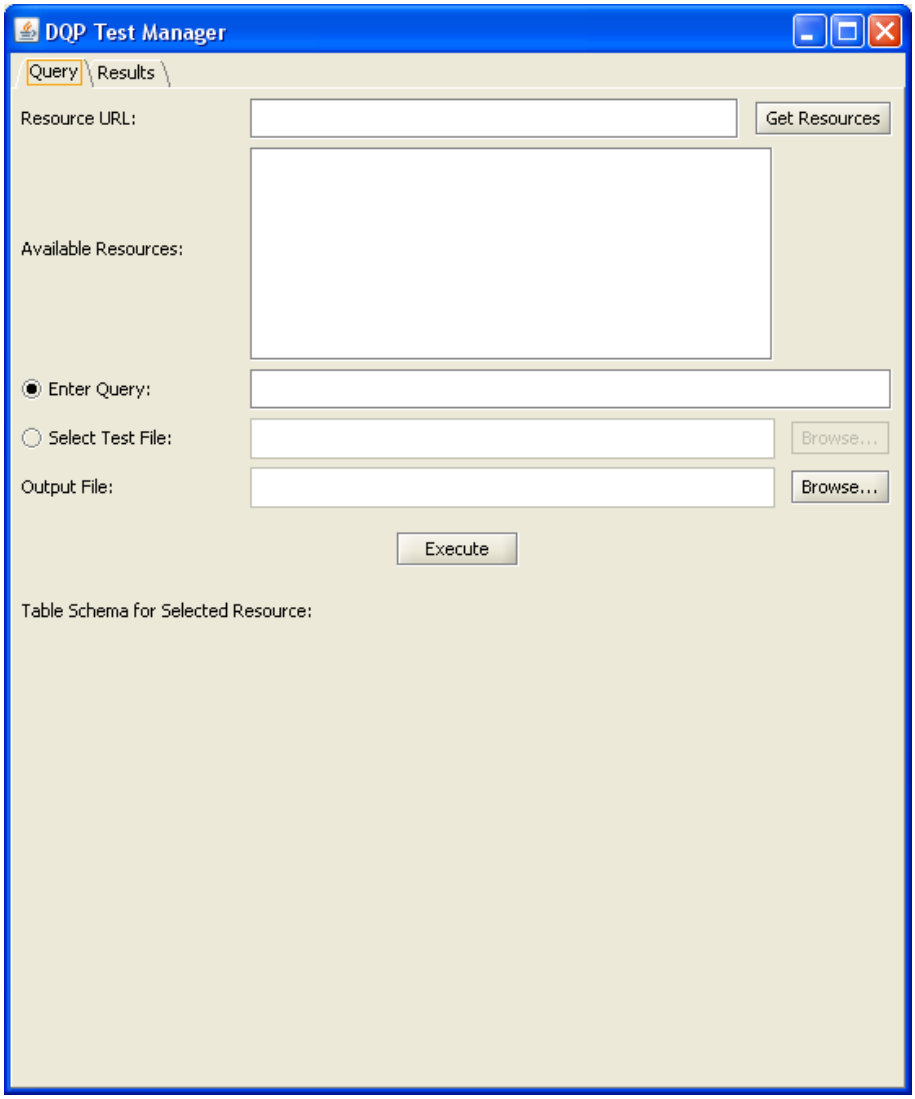

Figure 5.1: DQP Test Manager Main View

For these experiments, different datasets are generated arbitrarily using *DBMonster* [71] as mentioned in Section 5.1. Each data set and partition configuration is explained in the related experiment with details. The experiments are run for about 8-10 times and average execution times are calculated. The execution time of a single query may deviate 0-6 seconds from the corresponding average execution time. This deviation is generally very small with low cardinalities (at level of milliseconds) and increases by the data size due to possible increase in the rate of data flow over the network. Another issue to note here is that, test cases are run over a network that is actively in use, i.e. the network rate is changeable. In order to decrease the negative effect of the network rate in comparisons, the queries for a test case are

run successively, at a clear time of day when the network usage is average or low. Due to such possible network changes, the test cases are to be explored on their own.

5.3.1 Effect of Data Size on Performance

In Figure 5.2, *Partitioned-Scan-Query* and *Union All-Scan-Query* are compared to see whether there is a significant overhead incurred by the *UnionPartitions* operator. *Partitioned-Scan-Query* is executed on partitioned tables using the new operator, and *Union All-Scan-Query* is executed on the same partitions using *UnionAll* operator. The tables of partitions used in these queries are completely disjoint and distributed to four different physically separated computational resources. A total of 28 tables are generated for this experiment with names, users1, users2, users4, users8, users16, users32, users64 where the postfix number in the table names indicate the number of tuples in a single partition of the table in thousands. That is, for table with name *usersX*, four tables are created disjointly with names *usersX* and tuple sizes *X thousand*. So when a query is originated referring to a partitioned table, a total of *4X* thousand tuples are to be processed by the evaluators. The schema of the *Users* table is given in Table 5.1.

```
Partitioned-Scan-Query

SELECT userID, username, email, timestamp FROM AllCompanyUsersX;

Union All-Scan-Query

(((SELECT userID, username, email, timestamp FROM P1_usersX)
UNION ALL
(SELECT userID, username, email, timestamp FROM P2_usersX))
UNION ALL
((SELECT userID, username, email, timestamp FROM P3_usersX)
UNION ALL
(SELECT userID, username, email, timestamp FROM P4_usersX)));
```

Figure 5.2: Queries Used in Experiment 5.3.1

The average execution times are given in Table 5.4 for exploration. As it can be observed from Figure 5.3, the response times of both frameworks with and without the extensions are very close. It is also observed that the overall trend for scalability with data size of both operators is similar. This is also an indication that for strictly disjoint partitions the new operator can be a safe replacement for the UnionAll operator, presenting a much simpler query string as an advantage. But the replacements should be handled with care, since with

the increasing number of partitions, *UnionPartitions* may propose more overhead in building up binary *UnionPartitions* trees.

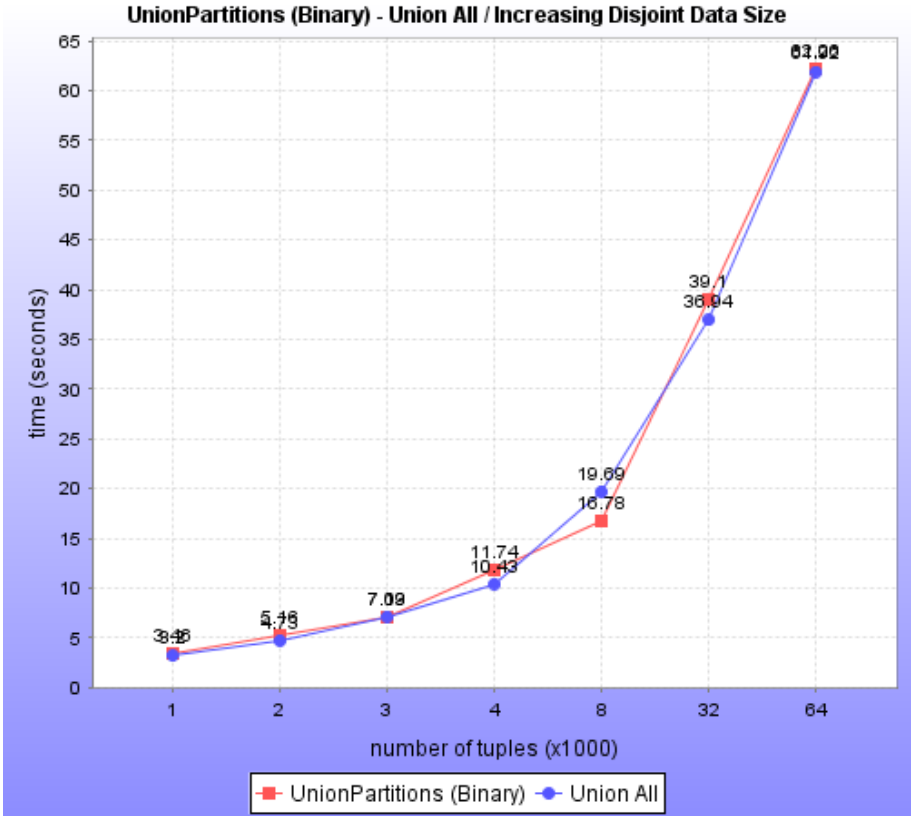

Figure 5.3: Query Execution Results For Experiment 5.3.1

Table 5.4: Execution Times for Experiment 5.3.1

	Number of Tuples (x1000)						
	1	2	4	8	16	32	64
Union All - Scan Query	3.20	4.73	7.13	10.43	19.69	36.94	61.92
Partitioned - Scan-Query	3.46	5.16	7.09	11.74	16.78	39.1	62.26

5.3.2 Effect of Increasing Overlapping Sections Among Partitions

This experiment is carried out to analyze the performance of the new operator with regard to varying proportions of overlapping data sections in the partitions. To achieve that *Scan-Overlapping-Query* given in Table 5.5 is executed several times with increasing overlapping

ratios between the partitions.

Table 5.5: Queries Used in Experiments; 5.3.2, 5.3.3 and 5.3.4

Query Name	SQL Query Text
Scan-Overlapping-Query	SELECT userID,userName,email,timestamp FROM CompanyUsers***X***;
Partitioned-FanIn-Query	SELECT userName,email,timestamp,userID FROM PartitionFanInTestOv***X***Part;
Partition-Configuration-Query	SELECT userName,email,timestamp,userID FROM SDTestSequentialBin***XY***Tuple;

In the query, X stands for the overlapping ratio between the partitions (e.g. for X=25, *CompanyUsers25* represents four tables with names *users25* on four different resources with overlapping sections of 25 %). The schema of the *Users* table is given in Table 5.1. The overlapping ratios between partitions are defined to range from a ratio of 25% overlapping to 100% overlapping (fully-replicated) by a step value of 25%. This experiment includes the elimination of outdated data based on the comparison of the *timestamp* field in each tuple. Therefore *Union* and *Union All* operators are not alternatives for the *UnionPartitions* operator because they do not provide the functionality to handle overlapping partitions. The *Union* operator would eliminate the duplicates comparing the whole tuple (based on a hash value calculated using the whole tuple), whereas *Union All* operator would keep all the data without any concern for duplicates.

The results in Figure 5.4 show that; the execution time for the query decreases as the overlapping ratios between the partitions increases. This decrease is an expected result regarding the decreasing number of processed tuples, due to the elimination of outdated tuples by the *UnionPartitions* operator, before they are delivered to the next upper operator in the query plan tree.

5.3.3 Effect of Increasing Operator Fan-In

This test is held in order to check whether the number of partitions of a table effects the performance of the binary *UnionPartitions* operator. In this test case, the number of records in

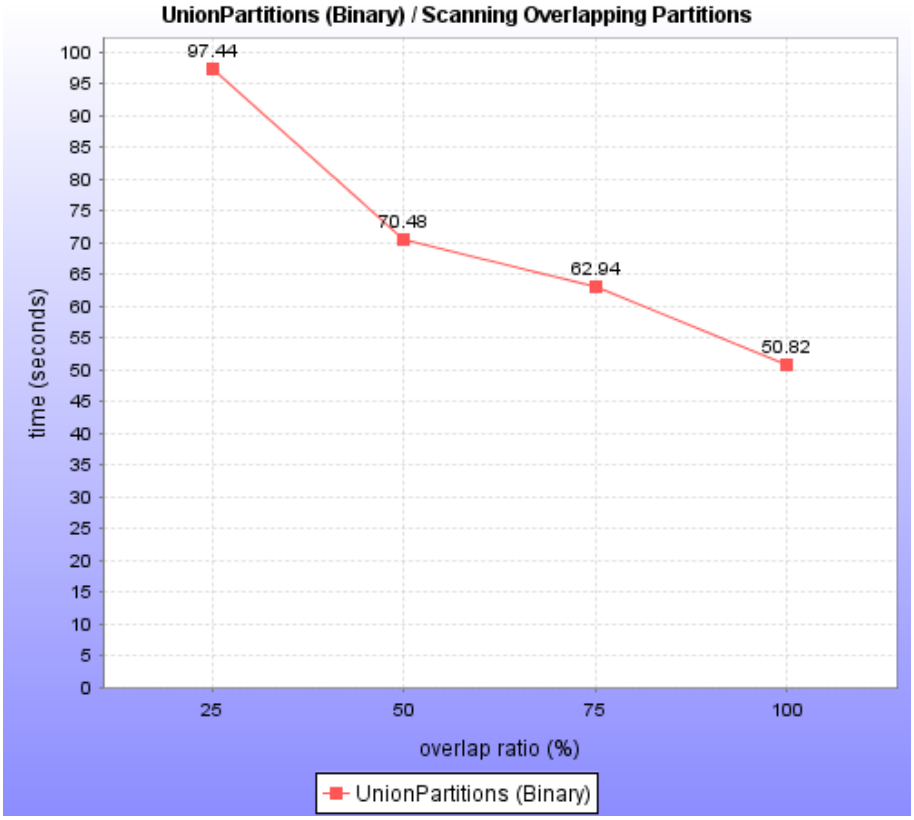

Figure 5.4: Query Execution Results For Experiment 5.3.2

each partition is adjusted based on the number of partitions participating in the query execution while the total data requested by the client is kept constant. *Partitioned-FanIn-Query*, used in this experiment is given in Table 5.5. The *X* in the query represents the number of partitions used in the query e.g *PartitionFanInTestOv5Part* refers to a table with 5 partitions. The execution times for queries over 1 to 8 partitions are observed. For this case, 36 tables are created and distributed to nodes T1, T2, T4 and T5. The number of records in each partition with change in partition count is given in Table 5.6.

The result of query executions is given in Figure 5.5. It is seen that, as the number of partitions increases, the execution time for the query also increases upto a point (here, 4 Partitions). After that, the execution time starts to drop as the number of partitions increases. First of all, this is is due to the fastly decreasing number of tuples pulled from each partition to keep the total data size constant. By decreasing the data pulled from each partition, the output production overhead of *UnionPartitions* operators in the tree is also decreased, since it waits

Table 5.6: Cardinality of Partitions when 50% Overlapping

Partition Count	Partition Cardinality
1	163.800
2	109.200
3	81.900
4	65.520
5	54.600
6	46.800
7	40.950
8	36.400

less for the completion of hashing left tuples. Secondly, it is seen that the query plan benefits from pipelined and independent parallelism more.

5.3.4 Comparison of Binary and Hybrid Trees

This test is held in order to check the effect of locating *UnionPartitions* operator by *HorizontalPartitionsOptimiser* using binary and hybrid approaches. For this test case, eight partitions are overlapped as binary. Each partition overlaps with only one other partition; such that 4 different internally overlapping, externally disjoint sets are produced.

The query used for this experiment is given in Table 5.5 with name *Partition-Configuration-Query*. The *XY* in the query represents the number of tuples used in the query e.g SDTest-SequentialBin***40***Tuple refers to a table with 8 partitions with 40 thousand tuples each. For this case, 168 overlapping tables (i.e. 21 tables for each partition) are created and for each of nodes; T1, T2, T4 and T5, 2 partitions are assigned.

The execution times for queries over 8 partitions with different configurations are observed and results are given in Figure 5.6. It is observed from the results that the hybrid approach is slightly superior to the binary approach, about 10 %. But the difference betwen the two seems to increase with the increases in data size.

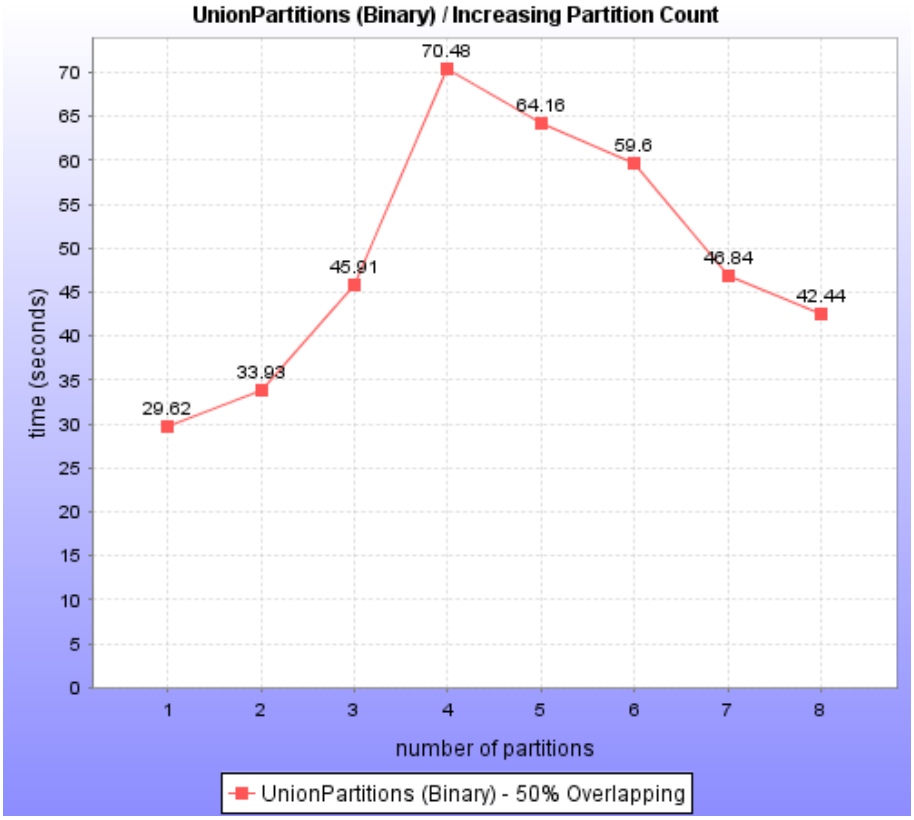

Figure 5.5: Query Execution Results For Experiment 5.3.3

5.3.5 Extension Scalability

This experiment is performed to find out whether the *UnionPartitions* operator hinder the existing scalability level of the OGSA-DAI DQP for more complex and realistic queries involving join operators. Since the *UnionPartitions* operator can be considered as a replacement for Union or UnionAll operators, it is expected that the behaviour of the framework will not change. However, here we present our test results to make sure that the reader is also informed about the fact that this theoretically expected outcome has been achieved by our implementation in practice.

The test is carried out by executing the *Partitioned-Join-Query* given in Figure 5.7. A total of 30 tables are generated for this experiment with names, usersX, projectsX and allocationsX, where X ∈ {'0', '25', '50', '75', '100'}. The postfix number in the table name represents the overlap ratio of the partitions for this table. Each partition contains 128 thousand tuples. That is, if the overlap ratio between two partitions is about 25% then, the primary keys of 32000

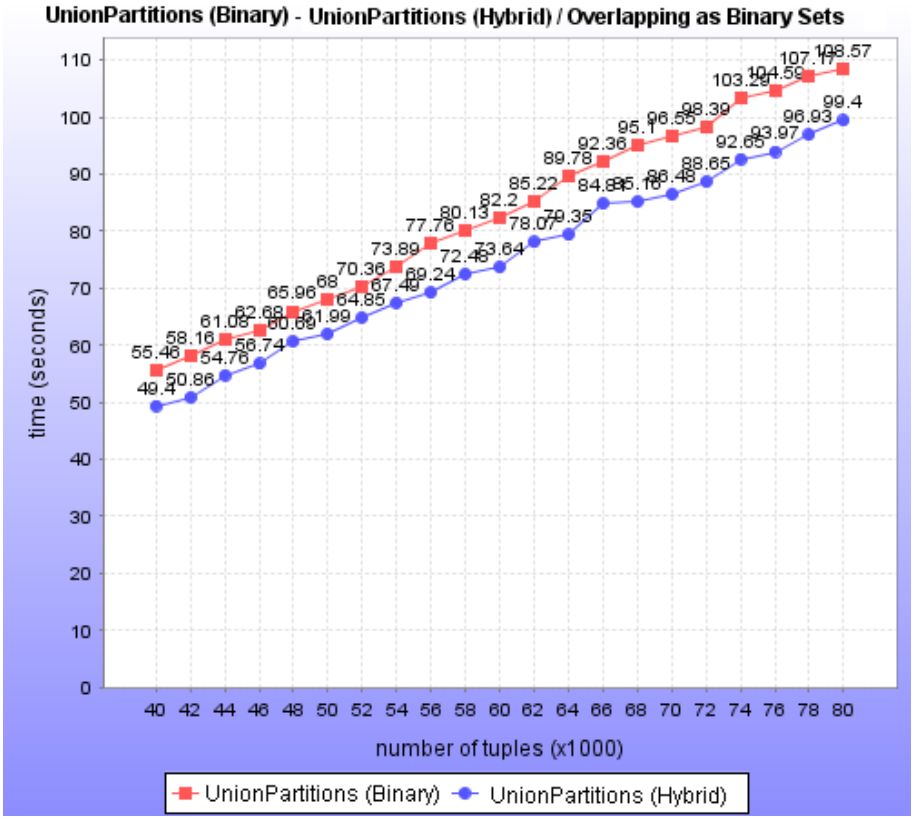

Figure 5.6: Query Execution Results For Experiment 5.3.4

tuples in each of the partitions are common and represent the same real-world object. The schemas that are common for all user tables, all project tables and all allocation tables used in this experiment are given in Table 5.7, Table 5.2 and Table 5.3 respectively.

The results of the executed query are given in Figure 5.8. As the results indicate, the execution

```
Partitioned-Join-Query

SELECT companyUsersX.userID, companyUsersX.userName, companyUsersX.email,
       companyUsersX.reminderType , companyUsersX.status,
       companyProjectsX.projectID, companyProjectsX.code,
       companyProjectsX.description, companyAllocationsX.toDate

FROM   companyUsersX,companyProjectsX, companyAllocationsX

WHERE  companyUsersX.userID=companyAllocationsX.userID and
       companyProjectsX.projectID=companyAllocationsX.projectID;
```

Figure 5.7: Query Used in Experiment 5.3.5

Table 5.7: Updated Schema for Users Tables

	Column Name	Column Type
	userID	bigint(20)
	userName	varchar(8)
	email	varchar(50)
	timestamp	timestamp
	reminderType	bigint(20)
	status	varchar(150)
Average Row Size (K)	378.6	

time for the join query decreases with a comparable trend to that of the previous query that did not involve a join operator. The slight fluctuation at the second ratio is likely to be caused by the increased rate of data flow over the network. It is evident that I/O cost is an important reason of such unexpected distortions. It is also known that the network is one of the primary costs for the queries in OGSA-DAI DQP because of the overhead of SOAP messages, which is explained in [12].

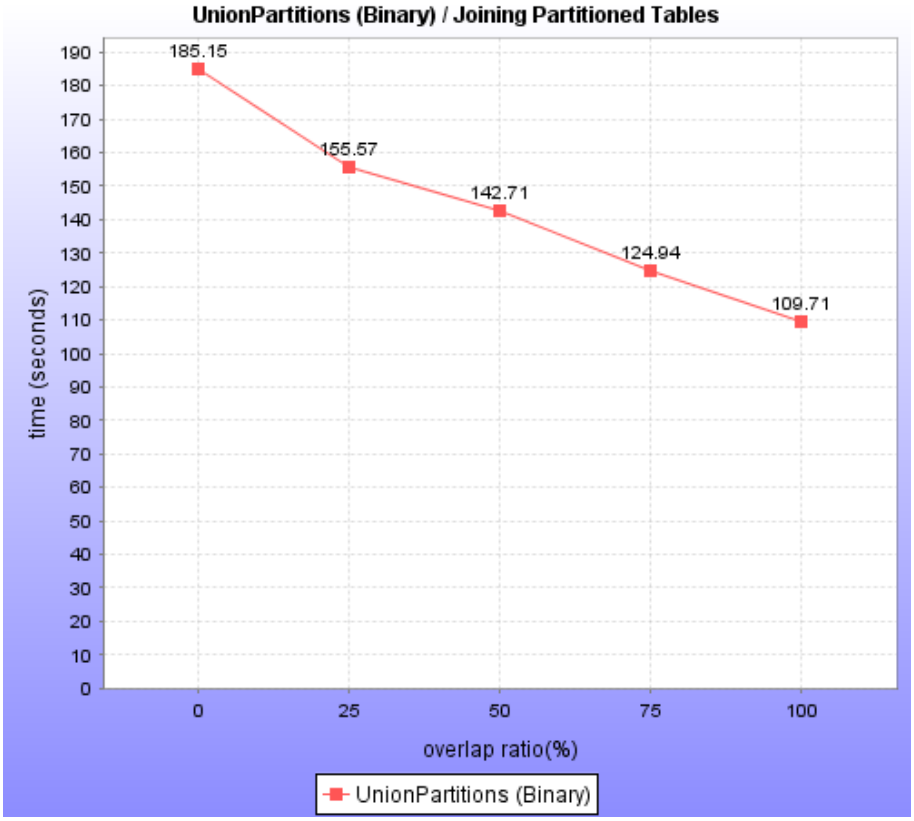

Figure 5.8: Query Execution Results For Experiment 5.3.5

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

This thesis targeted the problem of handling cases where a database was distributed to multiple independent administrative domains with the same data content initially (i.e. as replicas), but where parts of those multiple copies have evolved into overlapping partitions over time through independent data insertions carried out within each administrative domain.

As a solution, we presented an extension to the well-known data access and integration middleware, OGSA-DAI DQP, by incorporating the *UnionPartitions* operator into its algebra in order to cope with various unual forms of horizontally partitioned databases. The solution proposed extends OGSA-DAI DQP in two aspects:

1. A new operator type is added to the algebra to handle the specialized union of partitions with different characteristics. The new operator type is implemented and used by the corresponding new optimizer (*HorizontalPartitionsOptimiser*) with three different approaches; binary, strict n-ary and hybrid. The new operator has the following functionalities:

    - handles horizontally partitioned tables by supporting the user with an abstract table that stands for whole partitions
    - saves the user from writing long query expressions (involving joins and unions) for partitioned tables distributed over different systems with different databases
    - by being associative and commutative, the order of the application of the operator is not important in terms of the result. However the execution time is possibly affected.
    - gets the latest updated data for the overlapping partitions of the table

2. OGSA-DAI DQP Federation Description is extended to include some more metadata to facilitate successful execution of the newly introduced operator.

The approach presented in the thesis ensures that:

1. the modifications to the framework are done through explicitly specified extension points (i.e optimisers, activities), using well-defined mechanisms.

2. the extensions are non-disruptive to the internal behavioral characteristics of the framework, so they maintain its scalability level.

3. the extensions preserve the external APIs of OGSA-DAI DQP, so that the standard OGSA-DAI interaction patterns remain unchanged for external clients.

4. the new integration procedure is fully encapsulated into a new operator; thus the expansion of the integration capabilities is achieved through the extension of the algebra of OGSA-DAI DQP.

In order to illustrate the capabilities and performance of the extensions, several experiments are held and the results are reported. These results indicate that the new operator provides a convenient way of handling the mentioned requirements in regard to overlapping partitions, exhibiting reasonable performance characteristics.

The drawback of the presented approach is the need for the provision of additional metadata about the partitioning scheme of the data sources. Considering that this metadata is prepared once during the initial setup of the data integration environment, we argue that this requirement is relatively tolerable. Having said that, since the proposed metadata seeks to indicate some properties that can change over time (e.g. an overlapping partition may become disjoint) the need for keeping them up-to-date is an issue.

As future extensions, we aim at addressing these drawbacks and improving the algorithm of the new operator. First of all, to ensure that the metadata regarding the partition information is fully exploited, partitioning information and histograms could be produced and updated periodically in an automatic manner throughout query executions. As such, overlapping ratios between the partitions can be known beforehand, so that a more efficient algorithm for the ordering of the binary *UnionPartitions* operator can be developed. The algorithm then, would

consider using the most overlapping partitions (i.e. the partitions that have the highest amount of total overlapping ratio with the other partitions) in the first place to decrease the number of tuples flowing up the query processing tree. Clearly, such a capability would eliminate a part of the drawback stated above, but it requires new extensions to the activity workflows of OGSA-DAI services.

As a second future extension, the algorithm for the production of the bushy-like *UnionPartitions* operator trees can be improved to use more accurate data such that the output cardinalities of *UnionPartitions* operators can be calculated using several other parameters like overlapping ratios of inputs rather than only summing up. During the selection of inputs to a *UnionPartitions* operator, the decision of low-height trees to low-cardinality trees or vice versa should be decided based on some other heuristics. For example; rather than always preferring low-cardinality trees over low-height trees; a threshold determined based on some experiments can be employed to prefer low-height trees over low-cardinality trees in some cases.

Finally, since SOAP calls are not that efficient, the approaches here can be tried with better transfer protocols other than SOAP like in [75].

REFERENCES

[1] D. Kossmann, "The state of the art in distributed query processing," *ACM Computing Surveys (CSUR)*, vol. 32, no. 4, pp. 422–469, 2000.

[2] M. Antonioletti, M. Atkinson, R. Baxter, A. Borley, N. Hong, B. Collins, N. Hardman, A. Hume, A. Knox, M. Jackson, *et al.*, "The design and implementation of Grid database services in OGSA-DAI," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 357–376, 2005.

[3] OGSA-DQP, "Ogsa-dqp query compiler design internal document." `http://www.ogsadai.org.uk/documentation/ogsa-dqp_3.2/devdoc/QueryCompilerDesign.doc`, accessed on 01/01/2009.

[4] L. Bellatreche, "Horizontal Data Partitioning: Past, Present and Future," *Handbook of Research on Innovations in Database Technologies and Applications: Current and Future Trends*, 2009.

[5] A. Hameurlain, F. Morvan, and M. El Samad, "Large Scale Data Management in Grid Systems: a Survey," in *IEEE International Conference on Information and Communication Technologies: from Theory to Applications (ICTTA), Damas - Syrie, 07/04/2008-11/04/2008*, (http://www.ieee.org/), p. (electronic medium), IEEE, avril 2008. (Conférencier invité).

[6] S. Gatziu and A. Vavouras, "Data Warehousing: concepts and mechanisms," *Informatik (Zeitschrift der Schweizerischen Informatikorganisationen) 0*, vol. 1, 1999.

[7] G. Wiederhold, "Mediators in the architecture of future information systems," *Computer*, vol. 25, no. 3, pp. 38–49, 1992.

[8] Oracle, "Partitioning with oracle database 11g release 2." `http://www.oracle.com/technology/products/bi/db/11g/pdf/twp_partitioning_11gr2_2009_09.pdf`, accessed on 12/07/2010.

[9] Oracle, "Oracle9i database manageability." `http://www.oracle.com/technology/products/manageability/database/pdf/Oracle9iManageabilityBWP.pdf`, accessed on 12/07/2010.

[10] C. S. Mullins, "Achieving the five nines of database availability." `http://www.craigsmullins.com/dbta_038.htm`, accessed on 12/07/2010.

[11] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "The physiology of the grid: An open grid services architecture for distributed systems integration," in *Open Grid Service Infrastructure WG, Global Grid Forum*, vol. 22, pp. 1–5, Edinburgh, 2002.

[12] 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 Gener. Comput. Syst.*, vol. 25, no. 3, pp. 224–236, 2009.

[13] M. N. Alpdemir and D. Fitzgerald, "Experience on performance evaluation with ogsa-dqp," in *In Proceedings of the UK e-Science All Hands Meeting, 2005. CoreGRID TR-0113 12*, 2005.

[14] M. W. Ian Foster, Jens Vöeckler and Y. Zhao, "The virtual data grid: A new model and architecture for data-intensive collaboration," in *Conference on Innovative Data Systems Research*, 2003.

[15] I. Foster, A. Iamnitchi, M. Ripeanu, A. Chervenak, E. Deelman, C. Kesselman, W. Hoschek, P. Kunszt, H. Stockinger, K. Stockinger, *et al.*, "Giggle: A framework for constructing scalable replica location services," in *Conference On High Performance Networking and Computing*, pp. 1–17, IEEE Computer Society Press, 2002.

[16] S. Fiore, A. Negro, S. Vadacca, M. Cafaro, M. Mirto, and G. Aloisio, "Advanced grid database management with the greic data access service," in *Parallel and Distributed Processing and Applications*, vol. 4742/2007 of *Lecture Notes in Computer Science*, (Berlin), pp. 683–694, Springer Berlin / Heidelberg, 2007.

[17] "Globus toolkit software." `http://www.globus.org`, accessed on 12/07/2010.

[18] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor - a hunter of idle workstations," in *Proceedings of the 8th International Conference of Distributed Computing Systems*, vol. 43, pp. 104–111, 1988.

[19] W. Bell, D. Bosio, W. Hoschek, P. Kunszt, G. McCance, and M. Silander, "Project spitfire-towards grid web service databases," tech. rep., 2002.

[20] A. Umar, *Third Generation Distributed Computing Environments*. Nge Solutions, 2004.

[21] G. Graefe, "Query evaluation techniques for large databases," *ACM Computing Surveys*, vol. 25, no. 2, pp. 73–169, 1993.

[22] J. Smith, A. Gounaris, P. Watson, N. W. Paton, A. A. A. Fernandes, and R. Sakellariou, "Distributed query processing on the grid," in *GRID '02: Proceedings of the Third International Workshop on Grid Computing*, (London, UK), pp. 279–290, Springer-Verlag, 2002.

[23] D. Liu and M. Franklin, "GridDB: A data-centric overlay for scientific grids," 2004.

[24] S. Narayanan, T. Kurc, U. Catalyurek, and J. Saltz, "Database support for data-driven scientific applications in the grid," vol. 13, pp. 245–272, 2003.

[25] R. Fomkin and T. Risch, "Framework for querying distributed objects managed by a grid infrastructure," in *Data Management in Grids*, pp. 58–70, Springer-Verlag, 2006.

[26] J. H. Saltz, G. Agrawal, C. Chang, R. Das, G. Edjlali, P. Havlak, Y.-S. Hwang, B. Moon, R. Ponnusamy, S. D. Sharma, A. Sussman, and M. Uysal, "Programming irregular applications: Runtime support, compilation and tools," *Advances in Computers*, vol. 45, pp. 105–153, 1997.

[27] SweGrid, "Swegrid - the swedish grid initiative." `http://www.snic.vr.se/projects/swegrid`, accessed on 12/07/2010.

[28] NorduGrid, "Nordugrid middleware, the advance resource connector." http://www.nordugrid.org/middleware/, accessed on 12/07/2010.

[29] R. Fomkin and T. Risch, "Managing long running queries in Grid environment," in *On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops*, pp. 99–110, Springer, 2004.

[30] S. Venugopal, R. Buyya, and K. Ramamohanarao, "A taxonomy of data grids for distributed data sharing, management, and processing," *ACM Computing Surveys (CSUR)*, vol. 38, no. 1, p. 3, 2006.

[31] U. Srivastava, K. Munagala, J. Widom, and R. Motwani, "Query optimization over web services," in *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pp. 355–366, VLDB Endowment, 2006.

[32] T. Malik, A. S. Szalay, T. Budavari, and A. R. Thakar, "Skyquery: A web service approach to federate databases," in *Proc. 1st Biennial Conference on Innovative Database Systems Research (CIDR)*, 2003.

[33] V. Josifovski, P. Schwarz, L. Haas, and E. Lin, "Garlic: a new flavor of federated query processing for db2," in *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 524–532, ACM, 2002.

[34] S. B. Davidson, J. Crabtree, B. Brunk, J. Schug, V. Tannen, C. Overton, and C. Stoeckert, "K2kleisli and gus: Experiments in integrated access to genomic data sources," *IBM Systems Journal*, vol. 40, pp. 512–531, 2001.

[35] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, and K. Stocker, "ObjectGlobe: Ubiquitous query processing on the Internet," 2001.

[36] OGSA-DAI, "Projects using ogsa-dai." http://www.ogsadai.org.uk/applications/index.php, accessed on 12/07/2010.

[37] M. Antonioletti, M. Atkinson, N. Chuehong, A. Hume, M. Jackson, K. Karasavvas, J. M. Schopf, T. Sugden, and E. Theocharopoulos, "Grid enabling your data resources with ogsa-dai," 2007.

[38] M. Alpdemir, A. Mukherjee, A. Gounaris, A. Fernandes, N. Paton, P. Watson, and J. Smith, "An Experience Report on Designing and Building OGSA-DQP: A Service Based Distributed Query Processor for the Grid," 2003.

[39] M. Alpdemir, A. Mukherjee, A. Gounaris, N. Paton, A. Fernandes, R. Sakellariou, P. Watson, and P. Li, "Using OGSA-DQP to support scientific applications for the grid," 2005.

[40] M. Alpdemir, A. Mukherjee, A. Gounaris, N. Paton, P. Watson, A. Fernandes, and D. Fitzgerald, "OGSA-DQP: A service for distributed querying on the grid," 2004.

[41] "Ogsa-dai dqp." http://ogsa-dai.sourgeforge.net/documentation/ogsadai3.2.2/ogsadai3.2.2-gt/DPPart.html, accessed on 01/03/2009.

[42] B. Dobrzelecki, "Integrating distributed data sources with ogsa-dai dqp and views." EPCC, The University of Edinburgh - UK e-Science, All-Hands Meeting, www.ogsadai.ork.uk/archieves/presentations/ahm09-dqp-views.pdf, accessed on 01/08/2010.

[43] R. Schumacher, "Improving database performance with partitioning." http://dev.mysql.com/tech-resources/articles/performance-partitioning.html, accessed on 12/07/2010.

[44] S. Ceri, M. Negri, and G. Pelagatti, "Horizontal data partitioning in database design," in *SIGMOD '82: Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, (New York, NY, USA), pp. 128–136, ACM, 1982.

[45] O. M. T. and V. P, *Principles of Distributed Database Systems: Second Edition*. Prentice Hall, 1999.

[46] S. Agrawal, "Integrating vertical and horizontal partitioning into automated physical database design," in *In Proceedings of ACM SIGMOD*, pp. 359–370, ACM Press, 2004.

[47] L. Bellatreche, K. Boukhalfa, and P. Richard, "Data partitioning in data warehouses: Hardness study, heuristics and oracle validation," in *Data Warehousing and Knowledge Discovery*, vol. 5182/2008 of *Lecture Notes in Computer Science*, (Berlin), pp. 87–96, Springer Berlin / Heidelberg, 2008.

[48] K. Karlapalem, S. B. Navathe, and M. Ammar, "Optimal redesign policies to support dynamic processing of applications on a distributed relational database system," *Information Systems, Vol*, vol. 21, pp. 353–367, 1996.

[49] M. Unwalla, "A mixed transaction cost model for coarse grained multi-column partitioning in a shared-nothing database machine," *Information Systems*, vol. 19, no. 2, p. 193, 1994.

[50] H. Ma, K.-D. Schewe, and Q. Wang, "A heuristic approach to cost-efficient derived horizontal fragmentation of complex value databases," in *ADC '07: Proceedings of the eighteenth conference on Australasian database*, (Darlinghurst, Australia, Australia), pp. 103–111, Australian Computer Society, Inc., 2007.

[51] D.-G. Shin and K. B. Irani, "Fragmenting relations horizontally using a knowledge-based approach," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 872–883, 1991.

[52] S. Pramanik and S. Jung, "Description and identification of distributed fragments of recursive relations," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 6, pp. 1002–1016, 1996.

[53] G. Eadon, E. I. Chong, S. Shankar, A. Raghavan, J. Srinivasan, and S. Das, "Supporting table partitioning by reference in oracle," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 1111–1122, ACM, 2008.

[54] J. Chidambaram, C. Prabhu, P. Narasimha Rao, R. Wankar, C. Aneesh, and A. Agarwal, "A methodology for high availability of data for business continuity planning / disaster recovery in a grid using replication in a distributed database," in *TENCON 2008 - 2008 IEEE Region 10 Conference*, pp. 1 –6, 19-21 2008.

[55] R. Blankinship, A. R. Hevner, and S. B. Yao, "An iterative method for distributed database design," in *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*, (San Francisco, CA, USA), pp. 389–400, Morgan Kaufmann Publishers Inc., 1991.

[56] T. Wang, B. Yang, A. Huang, Q. Zhang, J. Gao, D. Yang, S. Tang, and J. Niu, "Dynamic data migration policies for query-intensive distributed data environments," in *Advances in Data and Web Management*, vol. 5446/2009 of *Lecture Notes in Computer Science*, pp. 63–75, Springer Berlin / Heidelberg, 2009.

[57] D. J. Reid, "Minimizing the response time of executing a join between fragmented relations in a distributed database system," *Mathematical and Computer Modelling*, vol. 25, no. 1, pp. 59 – 75, 1997.

[58] A. Segev, "Optimization of join operations in horizontally partitioned database systems," *ACM Transactions on Database Systems (TODS)*, vol. 11, no. 1, pp. 48–80, 1986.

[59] B. Gavish and A. Segev, "Set query optimization in distributed database systems," *ACM Transactions on Database Systems (TODS)*, vol. 11, no. 3, pp. 265–293, 1986.

[60] Y. Kambayashi and M. Yoshikawa, "Query processing utilizing dependencies and horizontal decomposition," in *SIGMOD '83: Proceedings of the 1983 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 55–67, ACM, 1983.

[61] D. Shasha and T. Wang, "Optimizing equijoin queries in distributed databases where relations are hash partitioned," *ACM Transactions on Database Systems*, vol. 16, no. 2, pp. 279–308, 1991.

[62] V. Ciglan and L. Hluchý, "Content Synchronization in Replicated Grid Database Resources," in *Third International IEEE Conference on Signal-Image Technologies and Internet-Based System, 2007. SITIS'07*, pp. 379–386, 2007.

[63] D. Dullmann, W. Hoschek, J. Jaen-Martinez, A. Samar, B. Segal, H. Stockinger, and K. Stockinger, "Models for replica synchronisation and consistency in a data grid," August 7-9 2001.

[64] M. Steinbrunn, G. Moerkotte, and A. Kemper, "Heuristic and randomized optimization for the join ordering problem," *The VLDB Journal*, vol. 6, no. 3, pp. 191–208, 1997.

[65] Y. E. Ioannidis and Y. C. Kang, "Left-deep vs. bushy trees: an analysis of strategy spaces and its implications for query optimization," *SIGMOD Record*, vol. 20, no. 2, pp. 168–177, 1991.

[66] G. Moerkotte, *Building Query Compilers (Under Construction) September 19, 2006*. http://pi3.informatik.uni-mannheim.de/ moer/querycompiler.pdf, accessed on 09/05/2010.

[67] A. Aljanaby, E. Abuelrub, and M. Odeh, "A survey of distributed query optimization," *The International Arab Journal of Information Technology*, vol. 2, no. 1, pp. 48–57, 2005.

[68] W. Scheufele, "Algebraic query optimization in database systems." PhD Thesis, Universität Mannheim, 15/02/1999.

[69] B. Kernighan and D. Ritchie, *The C programming language*. 1988.

[70] V. S., N. J., and B. J., "Maximizing the output rate of multi-way join queries over streaming information sources," in *Proceedings of the 29th VLDB Conference*, 2003.

[71] "Dbmonster," http://dbmonster.kernelpanic.pl/, accessed on 01/08/2010.

[72] I. Foster, "Globus toolkit version 4: Software for service-oriented systems," 2005.

[73] "Apache tomcat," http://tomcat.apache.org/, accessed on 01/08/2010.

[74] "Mysql," http://www.mysql.com/, accessed on 01/08/2010.

[75] 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)*, vol. 3, no. 2, pp. 1–30, 2008.

APPENDIX A

TEST DATA GENERATION FILE

The following file is used, during the generation of test data by 'dbMonster'; ;

```
<?xml version="1.0"?>
<!DOCTYPE dbmonster-schema PUBLIC "-//kernelpanic.pl//DBMonster Database Schema DTD 1.1//EN"
"http://dbmonster.kernelpanic.pl/dtd/dbmonster-schema-1.1.dtd">
<dbmonster-schema>
  <name>Test</name>
  <table name="test.users" rows="296132">
    <key databaseDefault="true">
      <generator type="pl.kernelpanic.dbmonster.generator.MaxKeyGenerator">
        <property name="columnName" value="userID"/>
      </generator>
    </key>
    <column name="userName" databaseDefault="false">
      <generator type="pl.kernelpanic.dbmonster.generator.StringGenerator">
        <property name="allowSpaces" value="true"/>
        <property name="excludeChars" value=""/>
        <property name="maxLength" value="8"/>
        <property name="minLength" value="0"/>
        <property name="nulls" value="0"/>
      </generator>
    </column>
    <column name="email" databaseDefault="false">
      <generator type="pl.kernelpanic.dbmonster.generator.StringGenerator">
        <property name="allowSpaces" value="true"/>
        <property name="excludeChars" value=""/>
        <property name="maxLength" value="50"/>
        <property name="minLength" value="0"/>
        <property name="nulls" value="0"/>
      </generator>
    </column>
    <column name="reminderType" databaseDefault="false">
      <generator type="pl.kernelpanic.dbmonster.generator.NumberGenerator">
        <property name="maxValue" value="127"/>
        <property name="minValue" value=""/>
        <property name="nulls" value="0"/>
        <property name="returnedType" value="integer"/>
        <property name="scale" value="0"/>
      </generator>
    </column>
    <column name="timestamp" databaseDefault="false">
      <generator type="pl.kernelpanic.dbmonster.generator.DateTimeGenerator">
        <property name="endDate" value="2004-01-01 02:00:00.07 +0200"/>
        <property name="nulls" value="10"/>
        <property name="returnedType" value="timestamp"/>
```

```
      <property name="startDate" value="1999-01-01 02:00:00.0 +0200"/>
    </generator>
  </column>
</table>
<table name="test.projects" rows="0">
  <key databaseDefault="true">
    <generator type="pl.kernelpanic.dbmonster.generator.MaxKeyGenerator">
      <property name="columnName" value="projectID"/>
    </generator>
  </key>
  <column name="code" databaseDefault="false">
    <generator type="pl.kernelpanic.dbmonster.generator.StringGenerator">
      <property name="allowSpaces" value="true"/>
      <property name="excludeChars" value=""/>
      <property name="maxLength" value="20"/>
      <property name="minLength" value="0"/>
      <property name="nulls" value="0"/>
    </generator>
  </column>
  <column name="description" databaseDefault="false">
    <generator type="pl.kernelpanic.dbmonster.generator.StringGenerator">
      <property name="allowSpaces" value="true"/>
      <property name="excludeChars" value=""/>
      <property name="maxLength" value="255"/>
      <property name="minLength" value="0"/>
      <property name="nulls" value="0"/>
    </generator>
  </column>
  <column name="timestamp" databaseDefault="false">
    <generator type="pl.kernelpanic.dbmonster.generator.DateTimeGenerator">
      <property name="endDate" value="2006-01-01 02:00:00.07 +0200"/>
      <property name="nulls" value="10"/>
      <property name="returnedType" value="timestamp"/>
      <property name="startDate" value="2004-11-01 02:00:00.0 +0200"/>
    </generator>
  </column>
</table>
  <table name="test.allocations" rows="0">
  <key databaseDefault="true">
    <generator type="pl.kernelpanic.dbmonster.generator.MaxKeyGenerator">
      <property name="columnName" value="allocationID"/>
    </generator>
  </key>
  <column name="userID" databaseDefault="false">
    <generator type="pl.kernelpanic.dbmonster.generator.ForeignKeyGenerator">
      <property name="columnName" value="userID"/>
      <property name="fastMode" value="false"/>
      <property name="nulls" value="0"/>
      <property name="tableName" value="test.users"/>
    </generator>
  </column>
  <column name="projectID" databaseDefault="false">
    <generator type="pl.kernelpanic.dbmonster.generator.ForeignKeyGenerator">
      <property name="columnName" value="projectID"/>
      <property name="fastMode" value="false"/>
      <property name="nulls" value="0"/>
      <property name="tableName" value="test.projects"/>
    </generator>
  </column>
  <column name="toDate" databaseDefault="false">
    <generator type="pl.kernelpanic.dbmonster.generator.DateTimeGenerator">
      <property name="endDate" value="2015-01-17 13:24:13.127 +0200"/>
      <property name="nulls" value="10"/>
```

```
        <property name="returnedType" value="timestamp"/>
        <property name="startDate" value="2009-02-01 02:00:00.0 +0200"/>
      </generator>
    </column>
    <column name="timestamp" databaseDefault="false">
      <generator type="pl.kernelpanic.dbmonster.generator.DateTimeGenerator">
        <property name="endDate" value="2009-01-17 13:24:13.127 +0200"/>
        <property name="nulls" value="10"/>
        <property name="returnedType" value="timestamp"/>
        <property name="startDate" value="2006-02-01 02:00:00.0 +0200"/>
      </generator>
    </column>
  </table>
</dbmonster-schema>
```

APPENDIX B

DQP TEST MANAGER

In order to show the capabilities of DQP Test Manager, which is used as a client in testing extensions performed on OGSA-DAI DQP System, example screenshots are given in following figures; B.1, B.2 and B.3.

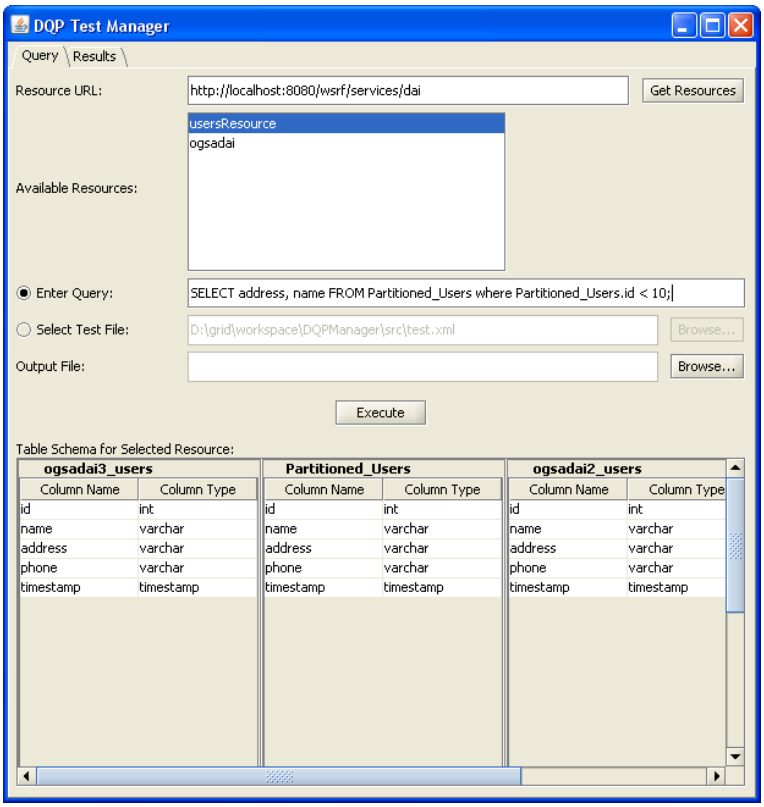

Figure B.1: Example Query Execution

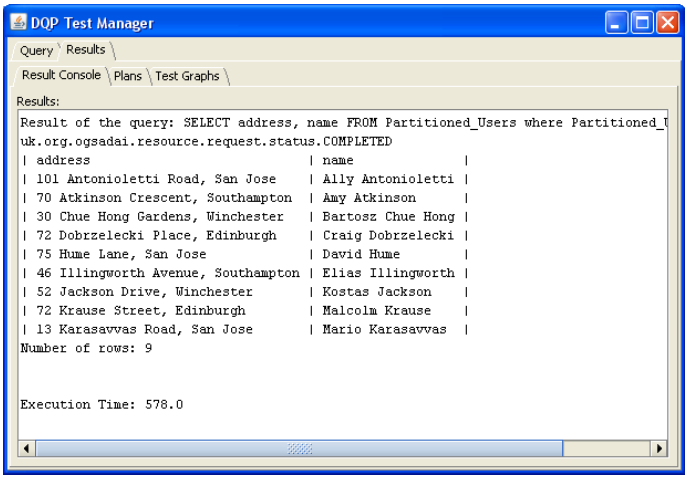

Figure B.2: Example Query Execution Result

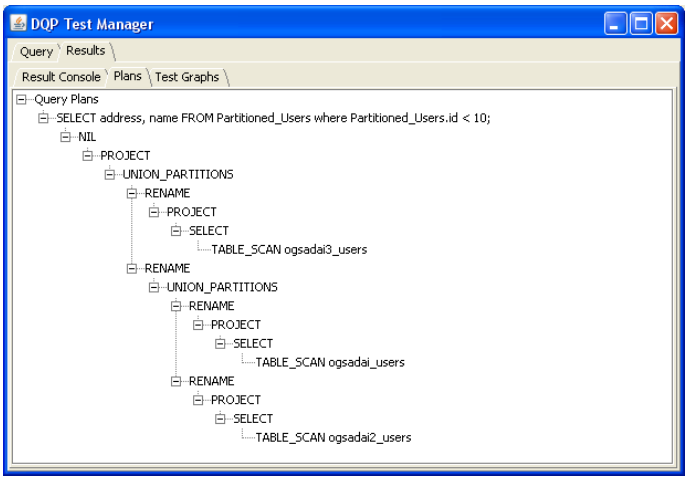

Figure B.3: Example Query Execution Plan

Although, single query executions over data resources, partitioned or not, is possible via DQP Test Manager; it is mainly written for executing queries one after another in order that measurements can be performed automatically. An XML file containing queries and some other parameters related with the executions is provided to DQP Test Manager. An example test file is provided in Figure B.4. An overview of the file is as explained. Several *graphs* can be defined in a test file. Each *graph* consists of several *query sets* which in turn contains several SQL *queries* related to an issue to be executed one after another. For each graph, in addition to providing parameters of *id*, *name*, *xAxisName*, *yAxisName* and elements of *query*

```
<graphs>

  <graph id ="1" name="Comparision for Union All and UnionPartitions (Binary)
                      - DisjointScanQuery / Increasing Data Size"
                        xAxisName="number of Tuples(x1000)" yAxisName="time (seconds)">

    <querySet id = "1" name="UnionPartitions(Binary) - DisjointScanQuery" warmUpCount="2">

      <query id = "1" execTime="10" sql="SELECT userID, userName, email, timestamp FROM SDTest1Tuple;"/>

      <query id = "2" execTime="10" sql="SELECT userID, userName, email, timestamp FROM SDTest2Tuple;"/>

      <query id = "3" execTime="10" sql="SELECT userID, userName, email, timestamp FROM SDTest4Tuple;"/>

      <query id = "4" execTime="10" sql="SELECT userID, userName, email, timestamp FROM SDTest8Tuple;"/>

      <query id = "5" execTime="10" sql="SELECT userID, userName, email, timestamp FROM SDTest16Tuple;"/>

      <query id = "6" execTime="10" sql="SELECT userID, userName, email, timestamp FROM SDTest32Tuple;"/>

    </querySet>

    <querySet id = "2" name="UnionPartitions(Binary) - DisjointScanQuery" warmUpCount="2">

      <query id ="1" execTime="10"
        sql="(((SELECT userID, userName, email, timestamp FROM userP1_users1)
        UNION ALL (SELECT userID, userName, email, timestamp FROM userP2_users1))
        UNION ALL ((SELECT userID, userName, email,timestamp FROM userP3_users1)
        UNION ALL (SELECT userID, userName, email,timestamp FROM userP4_users1)))"/>

      <query id ="2" execTime="10"
        sql="(((SELECT userID, userName, email, timestamp FROM userP1_users2)
        UNION ALL (SELECT userID, userName, email, timestamp FROM userP2_users2))
        UNION ALL ((SELECT userID, userName, email,timestamp FROM userP3_users2)
        UNION ALL (SELECT userID, userName, email,timestamp FROM userP4_users2)))"/>

      <query id ="3" execTime="10"
        sql="(((SELECT userID, userName, email, timestamp FROM userP1_users4)
        UNION ALL (SELECT userID, userName, email, timestamp FROM userP2_users4))
        UNION ALL ((SELECT userID, userName, email,timestamp FROM userP3_users4)
        UNION ALL (SELECT userID, userName, email,timestamp FROM userP4_users4)))"/>

      <query id ="4" execTime="10"
        sql="(((SELECT userID, userName, email, timestamp FROM userP1_users8)
        UNION ALL (SELECT userID, userName, email, timestamp FROM userP2_users8))
        UNION ALL ((SELECT userID, userName, email,timestamp FROM userP3_users8)
        UNION ALL (SELECT userID, userName, email,timestamp FROM userP4_users8)))"/>
      <query id ="5" execTime="10"
        sql="(((SELECT userID, userName, email, timestamp FROM userP1_users16)
        UNION ALL (SELECT userID, userName, email, timestamp FROM userP2_users16))
        UNION ALL ((SELECT userID, userName, email,timestamp FROM userP3_users16)
        UNION ALL (SELECT userID, userName, email,timestamp FROM userP4_users16)))"/>

      <query id ="6" execTime="10"
        sql="(((SELECT userID, userName, email, timestamp FROM userP1_users32)
        UNION ALL (SELECT userID, userName, email, timestamp FROM userP2_users32))
        UNION ALL ((SELECT userID, userName, email,timestamp FROM userP3_users32)
        UNION ALL (SELECT userID, userName, email,timestamp FROM userP4_users32)))"/>

    </querySet>

    <xAxisAttributes>
      <axisVal>1</axisVal>
      <axisVal>2</axisVal>
      <axisVal>4</axisVal>
      <axisVal>8</axisVal>
      <axisVal>16</axisVal>
      <axisVal>32</axisVal>
    </xAxisAttributes>

  </graph>

</graphs>
```

Figure B.4: Example Query Execution Test File

sets to be executed; a set of labels to be used for the x-axis should be supported. For each query set, other than *id* and *name* parameters and *query* elements; a *warmUpCount* parameter for specifying the count of execution before noting down the execution times, that is the count of throw-away executions should be provided. The *query* is the element where queries are defined. In addition to *id* and *sql* text parameters, each query is also associated with an execution count (*execTime*) parameter.