National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Version 1.0 (1997) 00000000

00000000 00000000 00000000

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

Interoperability in Multidatabases and its Application to Geographic Information Systems

by

Adel Mamhikoff

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

School of Computer Science

Carleton University
Ottawa, Ontario
May 12, 1993

© Copyright
1993, Adel Mamhikoff

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Author's Note: Author's Note

Author's Note: Author's Note

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-84097-8

Canada

Name ADEL MARINKLE

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

COMPUTER SCIENCE

SUBJECT TERM

0004

SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0327
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
General 0289
Ancient 0290
Linguistics 0291
Modern
Literature
General 0401
Classical 0294
Comparative 0295
Medieval 0297
Modern 0298
African 0316
American 0591
Asian 0305
Canadian (English) 0352
Canadian (French) 0355
English 0593
Germanic 0311
Latin American 0312
Middle Eastern 0315
Romance 0313
Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion
General 0318
Biblical Studies 0321
Clergy 0319
History of 0320
Philosophy of 0322
Theology 0469

SOCIAL SCIENCES

American Studies 0323
Anthropology
Archaeology 0324
Cultural 0326
Physical 0327
Business Administration
General 0310
Accounting 0272
Banking 0770
Management 0454
Marketing 0338
Canadian Studies 0385
Economics
General 0501
Agricultural 0503
Commerce Business 0505
Finance 0508
History 0509
Labor 0510
Theory 0511
Folklore 0358
Geography 0366
Gerontology 0351
History
General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science
General 0615
International Law and Relations 0616
Public Administration 0617
Recreation 0814
Social Work 0452
Sociology
General 0626
Criminology and Penology 0627
Demography 0938
Ethnic and Racial Studies 0631
Individual and Family Studies 0628
Industrial and Labor Relations 0629
Public and Social Welfare 0630
Social Structure and Development 0700
Theory and Methods 0344
Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture
General 0473
Agronomy 0285
Animal Culture and Nutrition 0475
Animal Pathology 0476
Food Science and Technology 0359
Forestry and Wildlife 0478
Plant Culture 0479
Plant Pathology 0480
Plant Physiology 0817
Range Management 0777
Wood Technology 0746
Biology
General 0306
Anatomy 0287
Biostatistics 0308
Botany 0309
Cell 0379
Ecology 0329
Entomology 0353
Genetics 0362
Immunology 0793
Microbiology 0410
Molecular 0307
Neuroscience 0317
Oceanography 0416
Physiology 0433
Radiation 0821
Veterinary Science 0778
Zoology 0472
Biophysics
General 0786
Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Cosmochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
General 0566
Audiology 0300
Chemotherapy 0992
Dentistry 0567
Education 0350
Hospital Management 0769
Human Development 0758
Immunology 0982
Medicine and Surgery 0564
Mental Health 0347
Nursing 0569
Nutrition 0570
Obstetrics and Gynecology 0380
Occupational Health and Therapy 0354
Ophthalmology 0381
Pathology 0419
Pharmacology 0572
Pharmacy 0382
Physical Therapy 0573
Public Health 0574
Radiology 0575
Recreation

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences
Chemistry
General 0485
Agricultural 0749
Analytical 0486
Biochemistry 0487
Inorganic 0488
Nuclear 0738
Organic 0490
Pharmaceutical 0491
Physical 0494
Polymer 0495
Radiation 0754
Mathematics 0405
Physics
General 0605
Acoustics 0986
Astronomy and Astrophysics 0606
Atmospheric Science 0608
Atomic 0748
Electronics and Electricity 0607
Elementary Particles and High Energy 0798
Fluid and Plasma 0759
Molecular 0609
Nuclear 0610
Optics 0752
Radiation 0756
Solid State 0611
Statistics 0463

Applied Sciences

Applied Mechanics 0346
Computer Science 0984

Engineering
General 0537
Aerospace 0538
Agricultural 0539
Automotive 0540
Biomedical 0541
Chemical 0542
Civil 0543
Electronics and Electrical 0544
Heat and Thermodynamics 0348
Hydraulic 0545
Industrial 0546
Marine 0547
Materials Science 0794
Mechanical 0548
Metallurgy 0743
Mining 0551
Nuclear 0552
Packaging 0549
Petroleum 0765
Sanitary and Municipal 0554
System Science 0790
Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis.

**Interoperability in Multidatabases and its Application to Geographic
Information Systems**

submitted by
Adel Mamhikoff
in partial fulfilment of the requirements
for the degree of Master of Computer Science

Director, School of Computer Science

Thesis Supervisor

Carleton University
April 30, 1993

Abstract

This work concerns the design and prototype implementation of the MultiDataBase Integrator. This system implements a federative client/server architecture that provides support for the three dimensions of a multidatabase system; namely, distribution, autonomy and heterogeneity. The MDBI proposes interoperability not only between DBMSs but also between DBMSs and/or GISs. Since the MDBI is a loosely coupled MDBS, its multidatabase language, the MDBI*SQL, is one of the core components of this system developed in this thesis. The MDBI's communication interface is built on top of TCP/IP Remote Procedure Calls protocol for Internets. This open system is not based on any particular data model or DBMS product. The current prototype focuses mainly on query interoperability and provides Gateways to Ingres, Oracle and heterogeneous GIS databases. It currently runs under UNIX on Sun and HP platforms. The working prototype is part of the Delta-X project being developed at the Geographic Information Systems Division at Energy, Mines and Resources Canada. The MDBI presents a testbed for further experimentation on other difficult issues in multidatabases such as multidatabase query optimization, semantic heterogeneity and transaction management.

Acknowledgments

The contribution of the people at the Geographic Information Systems Division at Energy, Mines and Resources Canada is gratefully acknowledged. I am eternally indebted to my parents who encouraged, inspired and believed in me. None of this would have been possible without their support. I would like to thank all my friends. I would also like to express my appreciation to my thesis supervisor Dr. Ekow Otoo. This work is dedicated to the new hopes for peace and prosperity in Lebanon.

Contents

Abstract	iii
Acknowledgments	iv
List Of Tables	viii
List Of Figures	ix
1 Introduction	1
1.1 Heterogeneous GIS Databases	3
1.2 Other Potential Multidatabase Applications	5
1.3 Scope and Organization	6
2 Characteristics of Multidatabase Systems	9
2.1 The 3D Model for Multidatabases	10
2.1.1 Distribution	11
2.1.2 Autonomy	12
2.1.3 Heterogeneity	14
2.2 A Taxonomy for Multidatabases	18
2.2.1 Loosely Coupled Federated Database Systems	20
2.2.2 Tightly Coupled Federated Database Systems	22
2.3 Schema Translation and Integration	25

2.3.1	Schema Translation	25
2.3.2	Schema Integration	28
2.3.3	DD/D and Auxiliary Schema	30
2.4	Multidatabase Languages	31
2.5	Query Processing and Optimization	37
2.5.1	Query Processing	37
2.5.2	Global Optimization	39
2.5.3	Query Translation	41
2.6	Transaction Management	41
2.6.1	Multidatabase Transaction Management Model	43
2.6.2	Restrictive Model with Standard Serializability	47
2.6.3	Multidatabase Serializability	49
2.7	Case Studies	49
2.7.1	Looseiy Coupled MDBSs	50
2.7.2	Tightly Coupled MDBSs	51
2.7.3	Other MDBSs	52
3	The MultiDataBase Integrator	53
3.1	MDBI Overview and System Layout	54
3.1.1	MDBI's Loosely Coupled Model	56
3.1.2	Client/Server Model for Remote Queries	58
3.1.3	Logical Topology of the MDBI	59
3.2	The MDBI*SQL Multidatabase Language	59
3.2.1	SELECT queries	60
3.2.2	COPYBD queries	64
3.2.3	EXPORT queries	65
3.2.4	REGISTER queries	67
3.3	GIS Interoperability with MDBI*SQL	67

3.3.1	Previous Work	68
3.3.2	Interoperability of GISs and/or DBMSs	69
4	Architecture of the MDBI	74
4.1	Remote Procedure Calls Model	75
4.2	The Virtual DataBase	78
4.2.1	Interactive MDBI*SQL User Interface	78
4.2.2	The Multidatabase Query Processor	79
4.2.3	DD/D and Auxiliary Schema Manager	85
4.3	The Remote DBMS Access Interface	88
4.3.1	The Data Structure Interface	89
4.3.2	The Remote Network Access Manager	91
4.3.3	The Network Interconnection Layer	93
4.4	Local DBMS Gateways	93
5	Current Prototype and Performance Model	96
5.1	Current Prototype	96
5.2	Performance Model	97
5.2.1	System Execution Protocol	99
5.2.2	Performance Related Issues	101
5.2.3	Analytical Model	103
6	Conclusion	113
6.1	Main Results and Contributions	114
6.2	Future Directions	116
	Bibliography	118

List of Tables

List of Figures

2.1	The 3D model for multidatabases	10
2.2	Taxonomy for multidatabases	18
2.3	ANSI/SPARC architecture for centralized DBMSs	19
2.4	Loosely coupled multidatabase system architecture	21
2.5	Tightly coupled federated architecture	24
2.6	Schema translation and integration	26
2.7	Query processing and optimization in multidatabases	38
2.8	Architectural model for transaction management in multidatabases	44
2.9	Distributed multidatabase transaction management	45
3.1	Architectural Layout of the MDBI	55
3.2	The loosely coupled model for MDBI	57
3.3	Distributed GIS interoperability with MDBI*SQL	70
4.1	DD/D Schema	86
5.1	Remote queries are executed in four steps	103
5.2	Queuing network for the MDBI running at three sites	105
5.3	Open network of queues for the analytical model of MDBI	106

Chapter 1

Introduction

Several data models for database systems have been proposed over the past few decades. These include the simple flat files used in data processing systems, Hierarchical, Network, Relational and the current proposals of object-oriented databases. For these data models, numerous database products have been developed and are now commercially widespread.

In the past several years different database products and data models have been adopted by different organizations and businesses. The same is true even within the same organization. Typically, these products, and especially the earlier ones, have been designed as centralized standalone DBMSs (database management systems) with limited or no access capabilities to other DBMSs. Added to this is the heterogeneity issues that arise between different data models and DBMS products.

Even with the use of the same data model, the presence of several, sometimes incompatible, DBMS products leads to serious heterogeneity problems. These problems include differences in the query languages and their extensions, differences in expressive power and performance, query processing and optimization techniques, transaction management (including concurrency control, commit protocols and recovery), and differences in hardware and software requirements and capabilities. Added to all these problems are data model and semantic heterogeneity when the

DBMS products belong to different data models.

This situation has led to what has come to be known as the islands of information problem. Over the years, massive amounts of data and applications have been accumulated by various organizations, and even within the same organizations, with limited or no interoperability. In this work, interoperability refers to users' ability to access and have an integrated view of data from multiple distributed database repositories. The islands of information problem has become even more widespread after the mergers and acquisitions trend of the 1980s. These DBMS systems and their applications are typically autonomous and must remain so. The result of all this is that these conglomerate businesses now operate a wide spectrum of DBMS products and applications that are essentially isolated.

Unfortunately, the well understood homogenous distributed DBMSs cannot be used to solve this problem. The reason is that although network distribution is an important component of the islands of information problem, it is by no means the only one. Heterogeneity and autonomy are equally important components of this problem.

To start with, setting up a distributed DBMS in an existing environment of databases is not feasible. Installing a distributed DBMS requires changes and disruption to the existing applications because they do not have the dichotomy of local versus global operations. Because this approach does not respect the independence of the existing databases, it will result in a complete change of the organizational structure for information management. The existing centralized DBMSs will have to be replaced by a distributed DBMS. On the other hand, it is both economically infeasible and virtually impossible to redevelop all these applications in a single DBMS environment. Even the choice of the unifying product and data model is not viable for the long term because of the continuous emergence of new data models and DBMS products.

Yet all this data should be somehow integrated to support company operations and management. *Multidatabases* have been proposed to deal with these problems. A multidatabase system is a software layer that provides distributed management

and transparent access to a collection of participating autonomous and heterogeneous database systems. Hence, multidatabases address the islands of information problem along its three orthogonal dimensions of distribution, autonomy and heterogeneity.

In a typical multidatabase environment, component DBMSs are neither aware of each other, nor even aware that they are participating in a multidatabase environment. The components cannot communicate to coordinate multidatabase activities. Therefore, it is up to the multidatabase to perform this function. Usually, each DBMS is 'wrapped' by a software layer. This is the multidatabase layer at that site. The local DBMSs can continue with their normal operations. The multidatabase layers are no more than any other application as far as the DBMSs are concerned.

Adding a layer of software above existing DBMSs is, hence, clearly different from the distributed DBMS approach. It allows the operations of existing applications to remain unchanged, preserves most of the organizational structure, supports controlled integration of existing databases, and facilitates incorporation of new applications and new databases.

1.1 Heterogeneous GIS Databases

Geographic Information System (GIS) databases are used to store and manipulate geographic data such as maps. Typically, GIS systems have a graphical interface, a storage subsystem, tools to digitize data into a map, and some analytical tools to perform analysis on maps such as compute the topology. There are several types of data that GIS products can manipulate [21,48]. These include raster data, structured and unstructured text and others. In our work, we concentrate only on vector data and attribute data.

Vector data represent maps as points, lines or polygons. Map features such as cities may be represented as either points or polygons depending on the map scale. Examples of lines features are utility lines, roads, rivers, elevation contours etc. Lakes, national boundaries, land use, soils, property lots etc. are all examples

of polygon features. Other levels such as feature codes, layers, coverages etc are also possible. These are methods of organizing data in a logical manner, and into manageable sizes. *Attribute data* store additional information about the feature such as the population of a city or the name of a country.

Different GIS products define and store their vector and attribute data differently. Typically, a GIS product will store its vector data in binary file(s) format unknown to the user. Some GIS products store their attribute data in ASCII files, others define their own DBMS, while others use a commercial DBMS. For example, ARC/INFO uses its own DBMS called INFO, while Caris uses Ingres (or Oracle¹ etc). Also note that ARC/INFO² can use Ingres³ or Oracle in addition to INFO. Hence, GIS products display a high degree of heterogeneity.

Clearly, GIS databases have all the symptoms that require a multidatabase solution. Users might be interested in accessing multiple GIS databases distributed over a network. We have already mentioned the heterogeneity issues. Autonomy is also present because the GIS databases and their data may belong to different organizations that want to retain control over their GISs as a condition to making them available to other users. Also, it is unlikely that rewriting the GIS software is a viable alternative to allow interoperability.

There is currently no interoperability between off-the-shelf commercial GISs. The only way to interchange data between two different GIS products is to manually download and upload the data using one of the standard interchange formats. These include such diverse formats as SIF (Integraph file format) and DLG (Digital Line Format). Most GIS products provide translators to at least one of these formats. However, having a way to seamlessly access data from multiple GISs, through ad-hoc queries in a high level non-procedural language (say SQL⁴), is a more flexible and appealing solution to this problem. In this work, a multidatabase is defined to

¹Oracle is a trademark of Oracle Corp.

²ARC/INFO, ARC and INFO are trademarks of Environmental Systems Research Institute.

³Ingres is a trademark of Ingres Corp. formerly known as Relational Technologies Inc.

⁴Structured Query Language

interoperate GISs in addition to the more traditional DBMS. Although the prototype development, in this thesis, is with respect to corporate databases and GISs, the scope of the techniques developed is by no means limited to these application areas only. The major thrust of the thesis is on the strategy adopted and has a wide applicability to other areas of scientific database and even file system applications.

1.2 Other Potential Multidatabase Applications

There are other examples of database applications that can benefit greatly from the multidatabase technology. While this work does not include these applications, we list them here for the sake of completeness.

Let's consider Information Retrieval Databases and Videotex Databases [43]. There are many well known information retrieval systems including Dialog⁵, CompuServe⁶, the Source⁷ and EasyNet⁸ in the U.S and Euronet/Diane in Europe. Minitel/Teletel⁹ provides comprehensive videotex services throughout France with over three million terminals and 4000 services in the late 1980s. These toll-services allow users to connect to a large number of databases. For example, EasyNet provides access to over 850 databases while Diane provides access to over 2000 databases.

These services define a query language for data access. Most of these systems use a variation of CCS (Common Command Set) language which is a standard in Europe. These systems have all the ingredients for multidatabases including the distribution component, the heterogeneity component and the autonomy component. The databases can be remotely accessed by users on a public network. The

⁵Dialog is a registered trademark of Dialog Information Services

⁶CompuServe is a registered trademark of CompuServe Inc.

⁷The Source is a service mark of Source Telecomputing Corp., a subsidiary of Reader's Digest Association, Inc.

⁸EasyNet is a registered trademark of Telebase Systems, Inc.

⁹Teletel is a service provided by the French telecommunications administration (PTT)

databases can actually reside on multiple sites and the data may be stored in different hardware, software and DBMS platforms. They are strictly autonomous. In France, most of the services provided over Minitel are commercial in nature and are provided by different business that clearly retain control over their databases.

Consider for example, the airline reservation service. Each airline has its own database that it makes available to the Minitel. Clearly, autonomy is a condition for the service since it is unlikely that any airline company will yield control over its data to another organization or a competitor. However, these systems are not multidatabases in a strict sense. While they have all the right ingredients, the services themselves lack an important feature of multidatabases: these systems provide access to only one database at a time. They make things easy for the users by providing menus. So a user can go through the airline company databases, one at a time, and choose a reservation. But, the user cannot specify an interdatabase query, a global query that will access multiple database. For example, the user cannot specify a query such as: "select the cheapest flight from Milan to Madrid over all airline companies". Clearly this is a feature that will appeal to a lot of users and potential customers of Minitel.

Another major area that can greatly benefit from multidatabase technology is multimedia. Multimedia includes hypertext, graphics, pictures, video, audio and hypermedia applications. Clearly, these applications have an inherent high degree of heterogeneity. There are currently ISO¹⁰ efforts to standardize SQL ADT (Abstract Data Type) packages [24] for multimedia data. Such efforts will greatly aid in multidatabase support for multimedia applications.

1.3 Scope and Organization

Many researchers and DBMS vendors have approached interoperability in multidatabases by imposing various restrictions on the multidatabase environment or by requiring rewriting of the components DBMS software. This work proposes the

¹⁰International Standards Organization.

MultiDataBase Integrator (MDBI). This system is a loosely coupled multidatabase backbone that provides support for the three dimensions of distribution, autonomy and heterogeneity in multidatabases. The following are the main proposals and results of the thesis:

1. Multidatabase interoperability using MDBI is not only between DBMSs, but also between DBMSs and/or GISs.
2. A multidatabase extension to SQL referred to as MDBI*SQL is custom tailored for GIS application development. It extends the qualifiers predicates proposed for multidatabase languages in the literature.
3. MDBI*SQL is used not only for DBMS interoperability, but also for distributed GIS interoperability. This proposal is unique to the MDBI project.
4. The MDBI is an open system not based on any DBMS product. In terms of architecture, the MDBI is composed of three subsystems: the *multi*-database manager, the network communication interface and the gateways to the local DBMSs and GISs.
5. Only the gateways change between different DBMS and GIS products. The Gateways maintain the full autonomy of component database systems. The Gateways do not require any special gateway support from the DBMS or GIS. They interface to these systems just like any other user application.
6. It is based on the client/server model using RPC for remote query executions. Each remote query is executed in two phases: the client phase where the sender is the client, and the server phase where the sender becomes the server to receive the results.
7. A prototype has been implemented. It supports global MDBI*SQL queries between Ingres and Oracle DBMSs distributed over a network. Translators to/from ARC/INFO and Caris GISs have also been built as an integrated part of the Delta-X GIS database project.

The rest of this text is organized as follows. Chapter 2 introduces the main characteristics of multidatabase systems. These characteristics include a taxonomy for multidatabases, schema integration, multidatabase languages, query processing and transaction management. Chapter 2.7 briefly discusses some of the typical prototype and commercial MDBSs. Chapter 3 proposes the MDBI for DBMS and GIS interoperability. This chapter also introduces MDBI*SQL, which is the multidatabase query language and the core of the MDBI system. Chapter 4 goes into the details of the architecture of the MDBI. Chapter 5 presents the current MDBI testbed and establishes an analytical performance model. Finally, Chapter 6 summarizes and concludes this work.

Chapter 2

Characteristics of Multidatabase Systems

A multidatabase system is a software layer that allows distributed and seamless access to multiple autonomous databases. Multidatabases are known under many other names. These names include *Federated* DBSs (database systems), *Interoperable* DBSs, *Heterogeneous* DDBMSs (distributed DBMS) and *Virtual* DBs (databases). All these names have been used interchangeably in the literature although each stresses a different feature of the multidatabase. Federated DBS, a terminology that is becoming more popular, refers to the autonomy component in multidatabases and hence stresses this feature. The term Interoperable DBSs refers mainly to highly autonomous and mainly retrieval systems. Whereas, heterogeneous DDBMS stress the heterogeneous nature of the component DBMSs. This is an older terminology and multidatabases were referred to as heterogeneous DBMSs in much of the earlier literature on this topic. Virtual DBs is a newer terminology. It emphasizes the database/site transparency in multidatabases. This term is used by Ingres/STAR [54] and Mermaid [65]. In this work, we will use all these terminologies to mean the same thing—a multidatabase.

Multidatabases are the most general case of database environments. The intuitive and logical structure of the ANSI/SPARC architecture prompted many researchers to consider ways of extending it to a more general environment. Before discussing MDBS taxonomies and functionalities, we present the possible ways in

Figure 2.1 The 3D model for multidatabases

which multiple databases can be put together for sharing by multiple DBMSs.

2.1 The 3D Model for Multidatabases

Multidatabases extend the traditional ANSI/SPARC architecture along three orthogonal dimensions. This classification organizes multidatabases as characterized with respect to the autonomy of the local systems, their distribution and their heterogeneity. We refer to this as the 3D model for multidatabases. This model is depicted in figure 2.1.

It should be clear from this model that a multidatabase is the most general case of a database environment. The origin of the diagram shown in the figure represents

a centralized database system. Homogeneous DDBMSs are represented along the distribution axis. Add to that autonomy and heterogeneity of components and the result is the general multidatabase environment.

2.1.1 Distribution

The *distribution* component refers to the distribution of the participating DBMSs and their data over a network. This distribution usually predates the MDBS, so it must cope with it. Data distribution also means that the data can be fragmented and replicated.

Fragmentation refers to the partitioning of data objects into smaller entities. For example, in the relational model there are clearly two alternatives for doing this: dividing the tables horizontally or vertically. Horizontal fragmentation partitions a relation along its tuples, while vertical fragmentation partitions a relation along its attributes. Of course, the two types of fragmentation can be mixed and applied repeatedly to a table.

Replication occurs when the same object is stored at multiple sites or databases. In the traditional homogeneous DDBMS environment, fragmentation and replication/allocation are performed in a top-down design process. These are usually performed to increase the performance of the DDBMS and the availability of its data. The decision to fragment and replicate is normally based on the applications that are expected to run on top of the DDBMS: while partitioning a table and distributing multiple copies of its fragments increase parallelism and hence the performance of retrieval queries, it may cause serious problems during updates.

A multidatabase environment is inherently based on a bottom-up approach of integrating and interoperating existing database repositories. Hence, fragmentation and replication are not solely a decision of the database administrator. Rather, parts of the data being integrated may represent replicas and fragments of some global data object that were previously stored and maintained separately. Indeed, there is no easy way of realizing that the data objects being integrated into a multidatabase environment represent partitions and replicas of the same thing. This gives rise to

some of the most difficult problems in multidatabases as we will see when we discuss semantic heterogeneity in Section 2.1.3 and schema integration in Section 2.3.2.

In addition to the network distribution, data may also be distributed among multiple databases. DBMSs typically define the concept of a database. A *database* is a named logical object which groups other data entities. For example, Ingres supports the concept of a database that groups a number of tables. Creating and maintaining databases is a decision made by database administrators to separate data, say by applications. A number of databases can reside on a single site. These databases may or may not be managed by a single DBMS. Even in the latter case, most centralized DBMSs will allow access to only one database at a time, so inter-database queries are not supported by such systems. Yet this is also a functionality that should be supported by MDBSs. Clearly, this is another added complexity of MDBSs over traditional DDBMSs.

2.1.2 Autonomy

The organizational entities that manage different DBs are often autonomous so that the local DBs are often under separate and independent control from the multidatabase. *Autonomy* refers to the distribution of control, rather than the distribution of data. Often participating DBMSs are willing to share their data only if they are allowed to retain control.

A component DBS participating in a multidatabase environment may exhibit several types of autonomy. A classification discussed in [16,27,67] includes three types of autonomy: design, communication and execution.

Design autonomy relates to the ability of participating DBMS to choose their own design with respect to the following:

1. The data being managed; i. e. the Universe of Discourse.
2. The naming of the data, and the representation of the data including the data model and query language.

3. The semantic interpretation of the data. This is related to the problem of semantic heterogeneity discussed in Section 2.1.3 below.
4. Constraints, including semantic integrity constraints and the serializability criteria, used to manage the data.
5. The functionality of the system and what operations it supports.
6. The association and cooperation with other systems (see association autonomy below).
7. The implementation of the query processor, transaction management algorithms, and the physical storage schemes.

This also means that the software of component DBMSs cannot be rewritten.

Communication autonomy requires that each of the individual DBMSs be free to make its own decision as to what information it makes available to the multidatabase environment. This implies that components decide how and when to respond to requests from the multidatabase.

Execution autonomy refers to the ability of participating DBMSs to execute local operations. Unlike the case of a DDBMS, component DBMSs can still be used locally as centralized systems. So local transactions are permitted on the DBMS despite its participation in a global environment. These local operations are not under the control of the MDBS and, in fact, the MDBS is not even aware of these operations. Hence, the MDBS cannot interfere with these operations. In short, local operations of the individual DBMSs are not affected by their participation in the multidatabase system.

Execution autonomy also implies that local and global operations are treated equally and global as well as local operations may be aborted. In addition, the MDBS cannot enforce an order of execution of global operations on a component DBMS. The external system will not be notified of the order in which the global commands are executed. Hence, the DBMS does not provide state information, such

as Local-Wait-For-Graphs to the MDBS. This causes serious problems in transaction management as we will see in Section 2.3.1.

In addition to the three types of autonomy mentioned above, [1] describes *association autonomy* which requires that each participator be free to associate or dissociate themselves from one or more multidatabase environments—or federations as we will sometimes refer to them.

In a nutshell, autonomy relates to the ability of components in a federation to retain control over their data and operations. Thus, the software of the DBMSs cannot be changed; the DBMSs may use any data models, query languages and transaction management schemes they want; participants have the choice to opt in or out of one or more federations; and local operations continue on the components without any control from the MDBS.

2.1.3 Heterogeneity

Local database systems in a multidatabase environment can be heterogeneous. This *heterogeneity* is mainly caused by the design autonomy requirement of DBMSs. Heterogeneity means that components participating in a multidatabase are different. These difference can occur at several levels.

Different DBMS products have different requirements for hardware and software support. This is referred to as *hardware and software requirement heterogeneity*. Hardware requirements includes the types of machines the DBMS may run on. This relates to the instruction set, the data formats and representation, and configuration of the hardware. Many operating system provide support for this type of heterogeneity. But this support is often limited to a set of hardware platforms that the operating system can run on.

DBMS software requirements mainly refers to the operating systems. DBMS products usually are not portable across different operating system environments. For example, the 4th Dimension DBMS is only available on MAC. Both Ingres and Oracle have different installations for UNIX and VMS platforms. This means that

some of the commands and features in these DBMSs may be different across different OS platforms. Not all DBMSs are available on all OS environments.

In addition to hardware and software heterogeneity, there is also communication heterogeneity. Different DBMSs may be distributed across different types of networks. There are several types of network heterogeneity. Communication lines can be different with respect to the physical links; for example, Ethernet, fiber optics, microwave etc. In addition, communication lines may operate in simplex, half-duplex or full-duplex mode. Transfer rates of different links are different.

There are also several types of transfer protocols including TCP [51], UDP [50] and VMTP [14]. These protocols operate differently and have various degrees of reliability. They also differ in terms of the size of network blocks they support. For example, TCP is presumed to be more reliable than UDP since it mandates acknowledgements for the messages it transfers. Also, network blocks in UDP can have up to 8Kbytes in size while TCP blocks can be as large as 4Gbytes [69].

One criterion for classifying networks is their geographic distribution. So networks are either WANs (wide area networks) or LANs (local area networks). Wide area networks can have different kinds of topologies including point-to-point irregular networks, star networks, tree or hierarchical networks, and meshed or complete networks. Examples of WANs are ARPANET, CSNET, MAILNET and EUNET [52]. These are also referred to as *Internets*. LANs are mainly buses and token rings. They are inherently faster than WANs because of their smaller geographical distribution. Often, LANs are connected to WANs via gateways and switches.

Fortunately, standardization efforts have solved many of the heterogeneity issues relating to networks. The International Standards Organization/Open System Interconnection (ISO/OSI) reference architecture [28] defines a 7 layer model. These layers address the issues presented above. The three lowest layers of this model—the physical layer, the data link layer and the network layer—are typically implemented on a communication subnet. The upper most four layers—the transport layer, the session layer, the presentation layer and the application layer—are available on many network programming tools including the Remote Procedure Calls protocol which

we discuss in section 4.1.

The application layer is dependent on the applications running on top of it. This is where most of the standardization efforts for open system interconnection are currently focused. For databases, ISO has been working on proposals for the Remote Database Access (RDA) protocol [24, 29].

Another type of heterogeneity is the *heterogeneity due to differences in the DBMSs*. An organization may have multiple DBMSs. Different departments and divisions within the organization may have different requirements and may select different DBMS products. DBMSs purchased over a period of time may be different due to changes in technology.

Different DBMSs may use different data models. The first data processing applications used flat file formats. Later, formal data models were introduced for DBMSs. These included the Hierarchical and Network (CODASYL) models. In the 1980s, database technology moved towards the higher level Relational model that supported ad hoc nonprocedural query languages. Currently, there are several proposals for an OO (object-oriented) data model for databases. No consensus has been reached yet among researchers as to what the model should be. While the OO model has not been standardized or even been well defined, several OODBMS have been built as prototypes and as commercial systems. So a multidatabase cannot disregard the OO model and OODBMSs.

Clearly, the difference in data models causes serious heterogeneity problems between DBMSs of different models. In fact, even within the same model, DBMSs made by different vendors can have substantial differences. The same holds true for different versions of the same product. There are several types of heterogeneities which are due to differences in DBMSs.

DBMSs may differ in many respects. The information structure supported may be different depending on the data model. For example, the relational model defines tables while the CODASYL model defines set types. The query languages of different DBMSs may be different. This is true even within the same data model. For instance, some relational DBMSs use SQL while others use QUEL. Finally, differ-

ent DBMSs may use different transaction management primitives and techniques, including concurrency control, commits protocols, recovery and deadlock handling protocols.

The last type of heterogeneity we discuss is *semantic heterogeneity*. Semantic heterogeneity deals with disagreement about the meaning, interpretation or intended use of the same or related data. This is a very difficult problem in multidatabases. It is poorly understood and there is not even an agreement regarding a clear definition of the problem [12].

Typically, DBMS schemas do not provide enough semantics to interpret data consistently. This is because their data models, specially in the relational case, do not capture enough in semantic information. Heterogeneity due to differences in data models contributes significantly to the difficulty of identifying and resolving semantic differences in data. Schema translation and integration discussed in section 2.3 deal with issues in semantic heterogeneity. The following example illustrates some of the difficulties in semantic heterogeneity.

Consider a relation **Course**(**grade**, ...) in a local database LDB^1 where **grade** describes student grades in the **Course** table and has the domain $\{A, B, C, D\}$. Consider another relation **Class**(**score**, ...) in a local database LDB^2 where the **score** denotes a normalized score on the scale of 0 to 10 derived by first dividing the weighted score of all exams on the scale of 0 to 100 in the course and then rounding to the nearest half-point. Clearly, $LDB^1.Course.grade$ and $LDB^2.Class.score$ are semantically heterogeneous. In this case, the heterogeneity is due to different precision of the data values taken by the related attributes. Attempting to compare student grades between the two databases can be very misleading. For instance, a *C grade* may correspond to a score of 61 to 75, which may not be possible to correlate with a *score* because both 73 and 77 could be represented by 7.5.

Figure 2.2 Taxonomy for multidatabases

2.2 A Taxonomy for Multidatabases

A taxonomy for multidatabases discussed in [61] and widely referred to in the literature is now introduced.

This taxonomy is based on the different approaches taken by different researchers in multidatabases. Since most of the work on this topic always assumes the presence of distribution and heterogeneity, this taxonomy mainly focuses on the autonomy dimension. It also addresses how the multiple database components are integrated into global schema(s), and who manages the schema(s).

Figure 2.2 shows the various alternatives based on autonomy and schema integration or the lack of it. An MDBS environment can either provide autonomy (at varying degrees) or not provide autonomy at all. In the case of total lack of autonomy, the system is referred to as a *nonfederated database system*. In this case, the participating DBSs are not autonomous so local operations are no longer allowed. Also, the components are fully integrated into a global schema much like the case

Figure 2.3 ANSI/SPARC architecture for centralized DBMSs

of homogeneous DDBMSs. This is an approach followed by UNIBASE [9]. In this text we will concentrate on the autonomous case only.

In contrast, a *federated database system* consists of component DBSs that are autonomous yet participate in a *federation*. They are willing to allow partial and controlled access to their data. There is no centralized control in a federated system because the component DBSs control all access to their data.

Federated DBSs can be classified as tightly coupled or loosely coupled based on their degree of autonomy and integration. Tightly coupled and loosely coupled can be differentiated in the way they extend the ANSI/SPARC architecture.

The ANSI/SPARC architecture is made up of three levels as depicted in figure 2.3. At the bottom most level is the internal schema. It describes the way data is physically stored by the DBMS. The intermediate level is the conceptual schema. This is a description of the whole database in the data model managed by the DBMS. Hence, concepts such as relational tables are defined at this level. The top most level is the external or view level. The optional schemas at this level define user views on the database; based on applications, security and other considerations.

2.2.1 Loosely Coupled Federated Database Systems

In a loosely coupled FDBS (federated DBS), the local schemas of component databases are not integrated into one (or more) global schema(s). Hence, it becomes the user's responsibility to create and maintain the federation. There is no control enforced by the federated system or its administrators. So federations users are the administrators of their schema. Loosely coupled FDBS can always support multiple federations.

The loosely coupled approach has been adopted by MRDSM [38-40] and SYBASE [43,66]. The central notion of the FDBS is the use of a *multidatabase language*. Section 2.4 is dedicated to multidatabase languages.

The ANSI/SPARC architecture is extended for MDBSs in the manner shown in figure 2.4. The levels of the loosely coupled multidatabase architecture are as described in [42]:

1. Internal Level: This level refers to the existing databases managed by DBMSs, each with its internal schema.
2. Conceptual multidatabase level: This is equivalent to the local DBMS conceptual schema in traditional ANSI/SPARC. This schema may be the actual local conceptual schema or a local external schema of the component DBMS. In the latter case, the actual conceptual schema is called an *internal schema*. The conceptual schemas at the multidatabase level may support different data models and may hide private data. If, on the other hand, some common data model (CDM) and language are required, for example, the relational model with SQL, it is the responsibility of each DBMS to respect it. This level also includes the definition of dependencies between subcollections of databases expressed as *dependency schemas*. These allow database administrators to specify and enforce interdatabase consistency despite the absence of a global schema. They may also define equivalence of attributes or domains, and so on. In practice these are enforced using triggers.

Figure 2.4 Loosely coupled multidatabase system architecture

3. External level: External database schemas can be constructed at this level. This is a notion similar to views in ANSI/SPARC.

Users access multiple databases directly at the multidatabase level, using the functions of the multidatabase language. Alternatively, users may access them through an external view, using either a multidatabase, or a database language if the external schema defines a single database.

Defining a multidatabase schema takes the form of schema importation. A user looks at the available set of local schemas, referred to in this case as *export schemas*, to determine which ones she or he would like to access. Local DBMS dictionaries may be used to help in this task. Next, the user imports the definitions of the schema objects using an interface or a facility in the multidatabase language. The new multidatabase schema is named and stored in a multidatabase level dictionary. It can be referenced or deleted at any time by the federation user.

Hence, loosely coupled FDBSs are inherently *dynamic*. Multidatabase schemas can be promptly created and dropped by the federated users. There are no pre-terminated processes for controlling the creation of such schemas.

Loosely coupled FDBSs acknowledge the fact that there are redundancies and discrepancies between the multiple databases. It is the responsibility of the users to understand the semantics of the objects and resolve semantic heterogeneity. This approach has the advantage of supporting multiple semantics depending on user applications and needs. Hence, it allows for higher flexibility in this respect. Indeed, database administrators may not have enough information to resolve these mismatches themselves. Therefore, the loosely coupled approach is very suitable for large numbers of very autonomous databases, used mainly for retrieval.

Note that loosely coupled FDBSs do not automatically support interdatabase consistency, even if the databases participate in the same external schema. It is the responsibility of the users to ensure that consistency is not violated. Therefore, the loosely coupled approach is not very suitable for supporting global updates.

2.2.2 Tightly Coupled Federated Database Systems

In a tightly coupled FDBS, it is the responsibility of the federation and its administrators(s) to create and maintain the federation, and actively control access to component DBSs.

The tightly coupled approach extends the three level ANSI/SPARC architecture into five levels as shown in figure 2.5.

1. **Local schema:** This is the conceptual schema of the component DBSs. As in the loosely coupled case, the local schema is expressed in the native data model of the DBMS, and hence different local schemas may be expressed in different data models.
2. **Component Schema:** These schemas are derived by translating local schemas into a *common or canonical data model* (CDM) for the multidatabase. This process is referred to as schema translation.

3. **Export schema:** An export schema represents a subset of a component schema that is available to the FDBS. Export schemas facilitate control and management of association autonomy by including access control information regarding their use.
4. **Federated schema:** A federated schema is an integration of multiple export schemas. It may also include information on data distribution that is generated when integrating export schemas although most systems separate this information into a separate *auxiliary schema*. Multiple federated schemas may be defined, each for a class of federation users as we will see further below.
5. **External schema:** An external schema defines a schema for (a group of) users and/or applications. This is done for customization purposes, since the federated schema can be quite large and complex. External schemas also provide additional integrity constraints and access control. An external schema can have a different data model than the unifying data model of the federated schema.

The creation of component, export and federated schemas are discussed in Section 2.3 which covers issues on schema translation and integration.

Tightly coupled FDBSs are better suited for traditional corporate environments. They provide more system control and better support for global updates. They also inherently provide for better location, distribution and replication transparencies; while loosely coupled FDBSs may require some sophistication from the federated users.

In figure 2.2, we see that there are two types of tightly coupled FDBSs: single federation and multiple federations. A tightly coupled FDBS is said to have a *single federation* if it allows the creation and management of only a single federated schema. This translates to having only one federated schema in figure 2.5 onto which all the export schemas are integrated. These systems provide the best support for global update capabilities. Having a single federated schema helps in maintaining semantic

Figure 2.5 Tightly coupled federated architecture

uniformity. However, this uniformity makes the problem of semantic heterogeneity harder to resolve.

If the multidatabase allows the creation and maintenance of multiple federations then it is referred to as a *multiple federated tightly coupled FDBS*. Having multiple federated schemas readily allows multiple integration of smaller sets of component DBSs which makes semantic heterogeneity more manageable. Constraints involving multiple component DBSs are, however, harder to enforce.

Variations on this taxonomy, or even other taxonomies based on the distribution or the heterogeneity dimensions are also possible. Several of these taxonomies are presented in [19, 44, 49, 53].

2.3 Schema Translation and Integration

Schema integration is only relevant to tightly coupled FDBSs. Schema translation and integration are often referred to simply as database integration. *Database integration* in a multidatabase environment is inherently a bottom up process. Database integration is performed in four main steps: translation of the local schemas into a common data model, defining the export schemas on the component schemas, integrating the export schemas into federated schema(s) and, finally, defining the external schemas. These steps are shown in figure 2.6.

Schema translation is the process of creating the component schemas. Export schemas can then be created by negotiation between a component DBA (database administrator) and the federation DBA. Schema integration is the process of creating federated schema(s) from the export schemas. The federation DBA creates and controls the federated and external schemas.

Database integration is a process that mainly considers data managed by DBMSs. A *pre-database integration* phase has to be performed when the participants include data not managed by DBMSs such as file system applications. There are two generic approaches to handle this type of data. The first approach is to migrate the files into a DBMS environment. This requires developing a component schema that describes the data in the files. The files can then be loaded into a DBMS. Also, the applications that access the data from the files will have to be modified so as to access their data from the DBMS. The second approach is to build a layer on top of these applications to give them DBMS-like features. Both these approaches are very expensive. Many developers of FDBSs elect to use migration.

2.3.1 Schema Translation

At the schema translation step, the local schemas of all component DBSs are translated into the same CDM. It is quite time consuming to write schema translators, but the choice of the CDM can facilitate the process.

Figure 2.6 Schema translation and integration

Schema translation is performed by the DBAs of each of the component DBSs. Two main approaches [61] can be followed. The first approach develops explicit mappings between the objects of source and the target schema. This approach is appropriate when a (class of) federation user(s) requires specific data structures in the target schema. The second approach develops general mapping rules to generate the target schema from the source schema.

The relational model is currently one of the most popular. So, practically, most commercial and prototype MDBSs use this model as the CDM. As we will also see in section 2.3.2, the CDM should be semantically rich. It should provide for abstraction and constraints so that semantics relevant to schema integration can be represented. Thus, semantic or conceptual data models are much more preferred for MDBS over traditional data models such as relational, network or hierarchical.

Many MDBS use the Entity-Relationship ER model. The ER model is a conceptual data model and captures the semantics of data by classifying the application domain into entities and relationships. The ER has also the advantage that it can be easily mapped into the popular relation model. Many extensions to the ER model (Extended ER or EER) have also been proposed. The Entity-Category-Relationship ECR model [20] is one such extension. In the ECR model, the concept of a *category* is introduced to represent generalization hierarchies and subclasses. A category is a subset of entities from an entity set. A category inherits the attributes of the object class (entity set or category) over which it is defined. Also, structural constraints on relationships are used to specify how entities may participate in a relationship. Cardinalities of constraints are not merely *one* or *many*, but are pairs of numbers (i_1, i_2) on each side of the relationship. Here, $0 \leq i_1 \leq i_2$ and $i_2 > 0$. The numbers mean that each entity of the object class may participate in at least i_1 and in at most i_2 relationship instances in the relationship set.

Continuing with the requirement for semantic data models, the object oriented paradigm appears to be the most appropriate basis for the CDM. In addition to its the highly semantic nature, the object oriented model provides for capabilities unseen in any other model. These include data encapsulation and abstraction, subclassing, subtyping and ISA hierarchies, inheritance, polymorphism, and complex objects. In addition, the concept of objects and methods (messages) closely matches the ever more popular client/server architecture in MDBSs.

In this section we assume that ER or some extension of it is used because they are closer to the relational model in addition to providing a higher degree of semantics.

Federation DBA use the component schemas to define the export schemas. However, the component DBA has the authority and control over what is included in the export schemas. The federation DBA is usually allowed to read the component schemas to help determine what data are available and where they are located and then negotiate for their access.

The *filtering processors* that relate the export schemas to their component schemas also control the access to the component DBSs. Access control necessi-

tates negotiation between the component DBAs and the federation DBAs [61].

2.3.2 Schema Integration

The process of integrating several export schemas into a single federated schema is called *schema integration* [3, 49]. In the case of single federation tightly coupled FDBSs, all the export schemas are integrated into one federated schema. While in the case of multiple federated tightly coupled FDBSs, several sets of (possibly intersecting) export schemas can be integrated into several federation schemas.

Identifying the components of the export schemas that are related to one another, selecting the best representation for the federated schema and, finally, integrating the components of each intermediate schema all make up the functionality of schema integration. Schema integration can be divided into four steps: preintegration, comparison, conformation, and merging and restructuring.

At the *preintegration* step, the integration rules are decided. For example, integration can be binary, where two schemas are integrated at a time, or *n*-ary, where more than two (possibly all) schemas can be integrated at the same time. Integration rules also include the identification of the candidate keys in each schema and the transformation rules between potentially equivalent sets of attribute domains.

Comparison involves identifying conflicts between the schemas. Conflicts may include naming (homonyms vs synonyms) conflicts, attribute domain conflicts and structural conflicts. Structural conflicts occur when the same object is represented as an attribute in one schema and as a entity set in another schema. They also occur when the same relationship set has different cardinalities in different schemas (e.g one-to-one vs many-to-many).

Once all these conflicts are identified, the relationships between the objects of the various schemas can also be identified. In the ER case, the type of the relationships will depend on whether the object is an attribute, an entity set or a relationship set. Two attributes can be *equivalent*, one can *include* the other or they may be *disjoint*. Determining such relationships can be computationally prohibitive if all pairs of attributes in all schemas have to be compared. Therefore, this process cannot

be automated. [60] suggests heuristics to identify only the relevant relationships (i.e other than disjoint) between a small subset of the attributes. Whereas in [62] *attribute equivalence classes* in ECR common data model are identified by the DBAs according to the real life meaning of the attributes.

Two entities may be *equal*, one entity may *include* the other, the two may *overlap* (i.e one may be integrated in the other), the two entities may be *disjoint but integrable*, or, finally, they may be *disjoint and nonintegrable*. Relationship objects may be similarly related. Here again, heuristic algorithms are used to identify pairs of entities and relationships that may be related by one of the first three relations [62].

Conformation is closely tied to comparison. At this step, also referred to as assertion, the conflicts identified above are resolved. Correspondences among the object classes and relationship sets across the different schemas are specified as assertions by the federated DBA. Conformation is the main vehicle for providing the semantic integration. Therefore, the assertions must be checked for consistency so that no group of assertions contradict the other.

Naming conflicts can be simply resolved by renaming conflicting objects. Structural conflicts can be resolved by transformations on attributes, entities and relationships [3]. In general, dependency or cardinality conflicts are resolved by opting for the more general one.

The *Merging and restructuring* step is the actual integration of the schemas. Related objects may be integrated by merging them into the same object or by placing them in an IS-A lattice with some objects being the generalization of other objects.

When the schemas are superimposed, the resulting intermediate schemas are analyzed and, if necessary restructured, in order to achieve several desirable qualities including completeness and correctness, minimality and understandability [3].

Many aspects are still missing in the methodologies for schema integration presented in the literature, including feasibility, performance and behavioral specifications.

The schema integration process is very hard if not impossible to fully automate. Schema integration tools have been proposed and are typically a form of an interactive browser that allows the database administrators to aid in integration process and resolve semantic conflicts. In [62] one such tool is presented. This tool uses the ECR model.

Once the federated schema(s) have been created, the external schemas can be created by negotiation between (a class of) federation users and the *federation DBA* who has the authority over what to include in each external schema. Often, an external schema defined on a federated schema requires a data model different than the CDM. The same schema translation methodologies can be applied for translating part of the federated schema into the external schema.

Existing applications need not be changed in a multidatabase environment. However, as old applications are modified, the component databases may be standardized and redundant data may be removed, unless required to improve performance or availability. New applications may be coded using an external schema defined on a federated schema. The FDBS evolves through gradual integration of a set of interrelated DBSs. It evolves as the new component databases are added and the existing ones are modified.

2.3.3 DD/D and Auxiliary Schema

It is important to keep in mind that while schema integration creates a logical unified view(s) of some or all component schemas, it does not necessarily create any actual data at the federation level. Rather schema integration results in meta data: the self-description of the federation.

During schema translation and integration, a *Data Dictionary/Directory* (DD/D) and *auxiliary schema* are constructed. These can later be used to aid in coordinating various activities by storing essential information on the federation. DD/D store meta-data about the schemas of the local DBSs as well as site/database allocation similarly to distributed DBMSs.

In multidatabases auxiliary schemas can be used to store information describing the following:

- How mappings between from/to participating schemas and the federated or external schema(s) can be performed.
- Additional information about schemas such as statistics and heuristics for query optimization.
- Schema-independent information such as tables and functions for unit/format conversions. For example, the formulas to convert between metric ground coordinates and longitude-latitude geographical coordinates.
- Various types of system information such as capabilities of each component DBMS, network addresses of each site and the communication facility to be used for communicating with these sites.

Finally, note that although loosely coupled MDBSs do not employ schema integration, DD/D and auxiliary schemas are still an important component of such systems. Of course, it becomes the responsibility of the federation users/DBAs to create and maintain these schemas. Creation of DD/D and auxiliary schema elements takes the form of schema importation in this case. Section 4.2.3 describes the DD/D and auxiliary database used by the MDBI.

2.4 Multidatabase Languages

The essential component of a multidatabase system is the language used to manage and manipulate the interoperable databases. Many researchers define a multidatabase as a federation of databases having a multidatabase language. Hence, a *multidatabase language* [41, 43] is an essential feature of an MDBS environment.

Schema integration allows the multidatabase language in tightly coupled FDBSs to be essentially the same as the centralized language of the CDM. This is because tightly coupled FDBSs provide a high degree of location, distribution and replication

transparencies. Hence, multidatabase languages and the extensions they provide are specially important for loosely coupled FDBSs that do not have the benefit of a global schema. Still, many tightly coupled FDBS also provide multidatabase extensions to their query languages.

Multidatabase languages provide the capability for defining federation or multidatabase level views. Also, these languages deal with problems identified in the discussion on schema integration. These include naming conflicts, data structure, type and unit/scale differences. Some systems define multiple semantics over the same data using multidatabase languages. Examples of multidatabase languages include MDSL [40], MSQL [41] and GSQL [30] and MALPHA [36]. The MRDSM project [37, 39] set much of the foundations of multidatabase languages.

Major issues in the design of such languages involve autonomy and redundancy. So, multidatabase languages must solve the problems of differences in data models, heterogeneity of names, value types and data structures discussed in Section 2.3.2. Also, data redundancy may cause serious update consistency problems.

Multidatabase languages should provide high level nonprocedural and ad-hoc manipulation of data much like centralized query languages. Multidatabase manipulations may involve interdatabase joins, updates spanning several databases, or even data interchange between participating databases. Multidatabase languages should also allow for redundancy of logical names, value types and data structures. These languages should be such that queries are as *reusable* as possible in the face of changes in the schemas of component databases. This is because changes to multiple autonomous schemas can be expected to be more frequent and less controlled than in centralized databases. Since there is no control over data, changes to schemas can occur at any time. It is unacceptable if all such changes require a reformulation of queries or applications.

Hence, multidatabase languages must support features not available in traditional database languages. The first and most common feature desired in multidatabase languages is the ability to qualify the names of data objects. Qualifiers include logical database names. Other desired features are the ability to allow for

autonomy while supporting cooperation between the administrators, especially over data definition. This means that data definitions can be made over several schemas in the multidatabase rather than requiring one individual definition for each of the component databases. Moreover, facilities are provided to support import and export of data definitions between databases.

SQL has become a common and popular query language for many research, prototype, and even commercial centralized systems. However, SQL lacks these fundamental multidatabase features discussed above. Therefore, extensions to SQL have been proposed to solve these problems. Before discussing MSQL, one of such extensions, we cite an illustration of MSQL from [41] to exemplify the motivation behind these extensions.

Consider the bank BNP (Banque Nationale de Paris) with three branches in Paris: Étoile, Nation and Opéra. Each of the branches as well as the main office has a database with the corresponding name. The databases contain the following information.

bnp: account(acc#, cl#, balance, brname)

etoile: account(balance, acc#, cl#, open_date)

nation: acc(type, acc#, cl#, balance, open_date)

opera: accounts(acc#, open_date, balance, cl#, category)

where **acc#** is the key of the relations.

A multidatabase, collectively called Branches, is declared by the command

```
CREATE MULTIDATABASE branches  
(etoile, nation, opera)
```

This is how schema importation is defined in loosely coupled MDBSs.

Assuming that, as in the usual practice in a bank, all branch accounts are also redundantly stored in the main office database; the balances of all branch accounts have to be copied to the main database each day. Assuming also that traditional

SQL is used, with name qualifier extensions, a query to perform this task can be formulated as follows:

```
DELETE FROM bnp.account
WHERE bnp.account.acc# = etoile.account.acc#
      OR bnp.account.acc# = nation.acc.acc#
      OR bnp.account.acc# = opera.accounts.acc#

INSERT INTO bnp.account
SELECT acc#, cl#, balance, 'etoile'
FROM etoile.account
UNION
(SELECT acc#, cl#, balance, 'nation'
FROM nation.acc
UNION
(SELECT acc#, cl#, balance, 'opera'
FROM opera.accounts))
```

This query does not provide data definition autonomy against changes in the schema definitions of the component databases. For example, if the names of any object in one of those schemas changes, this query also has to change. Moreover, if one of the branches closes (i.e., is deleted) or a new branch is added, this query also has to be changed to reflect the new situation.

Compare the above query to the one given below where more advanced multidatabase extensions are added to SQL.

```
USE bnp, branches
LET x BE branches.*
STORE bnp.account (* *)
SELECT *, NAME(x)
FROM x.acc%
```

here the **USE** clause defines the scope of the query to be **bnp** and **branches**. The **LET BE** clause defines the explicit *semantic* variable **x** whose range is the set of names of databases constituting **branches**. The **STORE** command replaces all tuples in **bnp.account** whose key matches the incoming tuples. The **(**)** clause means that the source attributes are mapped on the target attributes with the same names, regardless of their order in the table schemas. **NAME** is a built-in function that transforms a database name into a value to be stored in the **brname** attribute in **bnp.account**. Finally, **x.acc%** in the **FROM** clause is a compound semantic variable, where **x** represents the qualifier, **%** represents the wild character and **acc%** is any table name that starts with the prefix **acc**.

The **MSQL** formulation is much simpler than the previous **SQL** query. More importantly, it is much more reusable and provides a higher data definition autonomy. This statment need not be changed if new branches are added to the multidatabase, if object names changes as long as the **acc** prefix is used, or if new attributes such as **open_date** are added to **bnp.account**. Other related examples are also presented in [41].

The following features are desirable in a multidatabase language, in this case **MSQL**:

- Definition of a named multidatabases to be a composition of several component databases. The multidatabase name is used as the scope of data manipulation.
- Multidatabase level data definition statements including **CREATE**, **ALTER** and **DROP** of relations in different databases including the import of data definitions.
- Definition of units and precisions for data values in local schemas. This feature enhances the value type autonomy and enables automatic conversions between heterogeneous value types.
- Retrieval and update queries are called elementary queries in traditional **SQL**. These queries produce relations or update relations respectively. Multidatabase elementary queries obey the same rules as those for traditional

SQL except that relation names can be quantified. This makes it possible for retrievals and updates to span relations in several databases.

- The ability to perform in a single statement a relational operations, such as a projection on a single attribute, on a set of possibly heterogeneous relations.
- Implicit joins are used for manipulating databases with similar data but different vertical decompositions due to data structuring autonomy. An implicit join is a natural join performed by the system over domains that are defined to be compatible. It (re)constructs relations from their vertical decompositions. Like the universal relation interface, it allows for the same query formulation to run against multiple databases in which the same concepts are modeled through different relation decompositions. Thus, it allows for a single multiple query, where SQL would lead to several different queries.
- Dynamic attributes are defined for ad-hoc transformation between heterogeneous data values and a user defined basis. This features supports value type autonomy.
- Capability to update a dynamic attribute. MRDSM is the only system known to have dynamic attribute view update capabilities.
- New built-in functions such as NAME that allows transformation between data name and data value.
- Definition of virtual relation relations derived from multiple databases. Such relations are called multidatabase views.
- Definition of virtual databases as a named set of virtual relations.
- functions to deal with the problem of key conflicts between incoming and existing tuples in interdatabase queries for importing and exporting between databases. Also, support for single-statement transfers to or from several databases.

- Auxiliary objects like interdatabase triggers and equivalence dependencies between names in different databases used to specify equivalent and compatible domains and allow for implicit joins spanning several databases.

These features have been included in the MRDSM project which established their feasibility. Section 3.2 presents MDBI*SQL the multidatabase extension to SQL proposed for the MDBI project.

2.5 Query Processing and Optimization

Multidatabases provide federation users with the ability to combine data from different component databases in a single query. Multidatabase query processing requires different steps than distributed query processing [49], but the fundamental techniques are still very similar. In addition to that, the multidatabase environment requires query translation between the query language at the external or federated level and the individual query languages of the component DBMSs. Finally, optimization techniques and requirements in multidatabases are much more complex than those in the homogeneous distributed case. The query processing and optimization model in multidatabases is shown in figure 2.7.

In this model, queries submitted against the multidatabase are referred to as global queries. A *global query* may access data at the local database(s), at a remote database, at multiple databases or any combination thereof. Local queries can also be submitted at the individual component databases. However, these queries are submitted to the local DBMSs and hence the multidatabase query processor is not concerned, nor aware of these local transactions.

2.5.1 Query Processing

The site at which a global query is submitted is called the *control site* or the *query site*. The multidatabase query processor at this site is used to split and fragment a global query into *global subqueries*. Some query optimizations [25, 49], such as selection and projection optimization are performed at this step. Splitting the global

Figure 2.7 Query processing and optimization in multidatabases

queries is based on data distribution across sites using the data allocation information in the data dictionary component of the DD/D. Further fragmentation is performed, using the data directory component of the DD/D, so that the resulting subqueries are expressed on a single database at a single site.

Many MDBS query processing models translate the global subqueries into a *canonical query language* which is used in the global optimization tasks. This type of translation is performed at the initiating site. Canonical languages may facilitate the translation of the global subqueries into the native languages of the local DBMSs. For instance, general purpose translators can be constructed for a restricted subset

of relational query languages [57]. These translators take as input the syntax of the source and target languages in addition to the global subqueries and produce equivalent queries in the canonical language.

2.5.2 Global Optimization

Global query optimization in multidatabases is much more complicated than in the homogeneous distributed case. In a multidatabase environment additional complexities introduced to optimization due to heterogeneity and autonomy dimensions:

- There may be great differences between the speeds of the communication links, the speeds and work loads of the local processors and the nature of the data operations available of the local DBMSs.
- The cost of performing an operation may be different on different component DBMSs. Because of the autonomy of component DBMSs, the cost of performing the operation may not be known or may only be approximately known. This cost may also vary with system load.
- The local optimization capabilities of each component DBMS may be quite different.
- Different DBMSs provide different system and database operations. These operations include the ability to create temporary relations, algebraic (select, project and joins) operations, aggregate operations, automatic removal of duplicates etc.
- Substantial overhead in data translation between different DBMSs, specially if they are based on different data formats.
- Multidatabase languages, as we have seen in section 2.4, introduced new capabilities whose optimization is still an open area of research.

These prevent uniform treatment of queries across multiple component DBMSs, unlike the case of DDBMSs. In addition, the *total cost* [25, 49] function is more complex as it contains many more components and constraints for the items mentioned above.

The global query optimizer at the control site is sometimes referred to as the Global Data Manager (GDM). The components of the MDBS layers at the local DBMSs that receive the global subqueries and submits them to the DBMS for execution are called Local Data Interfaces (LDIs). Global optimization involves a trade-off between the amount of work done by the GDM and the complexity of the LDIs, and between the amount of communication and processing done by different component DBSs. MDBS query optimization can follow one of the three design approaches [61]:

- Simple LDIs and GDM: This assumes that global subqueries were generated to be the smallest possible local queries. For example, in the relational model, a separate subquery is generated for every table in the scope of a global query instead of every database/site. This results in a heavy workload for the GDM which merges the results and signification communication between the GDM and the LDIs.
- Medium complexity GDM and LDIs: This follows closely the model presented in this section. It reduces the GDM workload and communication between the GDM and the LDIs
- Complex GDM and LDIs: The GDM generates efficient programs that involve the participation of LDIs in the global optimization. Partial results may be sent back to the GDM as well as other LDIs. LDIs support functionalities including sorting, duplicate removal, and handling and merging temporary files. This approach results in further reduction of communication and GDM. This approach is followed by multidatabase systems such as Mermaid [13].

In addition to global optimization, the LDIs may use the facilities and features of the native query language and physical database organization of the component DBMSs to perform further "local" optimizations.

2.5.3 Query Translation

Before the LDIs at the participating sites can submit a global subquery (possibly expressed in the canonical query languages) for execution against a local DBMS, they must translate the subquery into the native (or target) language of the local DBMS. The auxiliary schema can be used to aid in this process as shown in figure 2.7. The same principles can be used to translate between external schema and federated schema query languages if applicable.

Different DBMSs and data models have different query languages. In general, query languages can be categorized as either procedural or nonprocedural. An interesting issue is the translation between the two types of languages [11, 15].

Loosely coupled MDBSs always support multiple federations. These may use different (multidatabase) query languages. Even in the case of a tightly coupled MDBS environment, the various federations and external schemas may also support different query languages. Hence, building "bilateral" translators in such environments can be very costly: $O(n^2)$ for n languages. In contrast, using a standard intermediate (or canonical) language reduces the number of translators to $O(n)$. This also means that the addition of a new DBMS to the multidatabase environment will only require a translator between the new query language and the canonical query language, instead of potentially $O(n)$ new translators.

2.6 Transaction Management

Recall from Section 2.1.2 that both local and global transactions are permitted on the participating databases. *Local transactions* are transactions that are executed by a local DBMS outside any control from the multidatabase. By definition, local transactions are transactions submitted by the local users on the local DBMSs. Hence, these transactions can access only data located at a single database of a single site. Global transactions are submitted to and executed under the control of the MDBS. Global transactions may access data located at several databases and several

sites. The basic problem is hence supporting concurrency between transactions given that the MDBS does not know about the local transactions.

The following is a summary of the main assumptions typically made in a multidatabase transaction management model.

- No changes can be made to the local DBMS software. This is the requirement of the component DBMSs for strict autonomy.
- Local DBMS cannot communicate to synchronize the execution of global transactions. This has to be done by the MDBS layers running at each site.
- Component DBMSs are responsible for the atomicity of the local transactions, but do not provide any state information to the MDBS.
- Local DBMSs are not able to distinguish between local and global transactions.
- A local DBMS cannot participate in the execution of a global atomic commit protocol, e.g., two phase commit (2PC).

Indeed, these assumptions make the MDBS environment very general. Furthermore, the difficulties encountered in multidatabase transaction management are largely due to these assumptions.

This leads to two conflicting requirements. On the one hand, autonomy requires that the global transaction management functions be performed independently of the local transaction execution functions. Therefore, the transaction managers of the individual DBMSs cannot be modified to accommodate global updates. On the other hand, heterogeneity creates difficulties in making uniform assumptions about the functionalities provided by component DBMSs. Further complications occur when the different component DBMSs (and the MDBS) use different concurrency control mechanisms.

Nowhere in multidatabases does the autonomy of the component DBMSs present more problems than it does in supporting updates. The challenge becomes to permit concurrent global updates to the underlying databases without violating their autonomy. It is clear, hence, that transaction management is one of the most difficult

issues in multidatabases. Indeed, transaction management is probably one of the major open questions in this field. This is underlined by the fact that none of the research, prototype or commercial MDBSs has been able to support global update capabilities. In fact, all the solutions published so far either impose various, often unrealistic, assumptions and restrictions on the MDBS environment, compromise autonomy by requiring major rewriting to the DBMS transaction managers, deviate from the traditional serializability theory, or even use a custom tailored approach to transaction management based on specific applications [17, 43, 49, 61, 63, 66].

2.6.1 Multidatabase Transaction Management Model

The architecture of the transaction management model in multidatabases is shown in figure 2.8. As stated before, there is an MDBS layer on top of each participating DBMS.

Usually, the local DBMSs are assumed to provide no more than the functionality to receive “user” transactions, execute to termination (either commit or abort) and report the results back to the user. This means that the global transaction manager of the MDBS cannot obtain any state information about the transaction managers of the local DBMSs such as a local wait-for-graphs. From the perspective of the local DBMSs, the MDBS layer is just another “user” from which they receive transactions and present results.

A *server* process is generated by the global transaction manager at some local site to act as an agent for a global transaction at that site. Hence, in figure 2.8 transaction T_1 has a server at $DBMS^l$ and another server at $DBMS^k$. Each time an operation of a global transaction is scheduled and submitted for execution, it is eventually received by the server. Results of the operation execution by a local DBMS are reported to the global transaction manager by the server. In addition, the server also needs to interact with the global transaction manager after a failure has occurred.

Each server has a *log* which it uses to keep track of the changes made by the global (sub)transaction it is responsible for. The log is kept in stable storage. The

Figure 2.8 Architectural model for transaction management in multidatabases

server, hence, updates the log before each update to the local database. Therefore, the server log is a *write-ahead log* that contains only *redo* records.

The global transaction manager sends each of the operations of a given global transaction to the server process responsible for that global (sub)transaction at the corresponding site. In such an environment, a series of conditions have been defined that specify when a global transaction can safely update a multidatabase system [26, 27]. These conditions are helpful in determining the minimal functionality required by the various transaction managers. The first condition is to have the individual DBMSs guarantee local synchronization as stated previously.

The second condition requires that the local transaction managers maintain the relative execution order of the subtransactions determined by the global transaction

Figure 2.9 Distributed multidatabase transaction management

manager. Since the global transaction manager is no more than an application running on top of the local DBMSs, the way to guarantee that this condition holds is to require that the global transaction manager awaits the result of one operation of a subtransaction before submitting the next to maintain the ordering of the operations. That is, no operation of a global transaction (except the first) can be submitted for execution until the global transaction manager receives a response that the previous operation of the same transaction has completed.

The third condition relates to the fact that the global transaction managers of the MDBS layers at each site are distributed. In terms of figure 2.8, the global transaction manager shown is that of the MDBS layer to which transactions T_1 , T_2 and T_3 are submitted. If the local databases are distributed over several sites (as opposed to having multiple databases at the same site), then the global transaction manager at the initiating site would have to coordinate with the global transaction managers of each of the MDBS layers at the participating sites. Hence, communication between the global transaction manager and its servers would also have to be done through the MDBS layers at those sites. This situation is more accurately described in figure 2.9.

Recall from section 2.5 that the query processor splits a global query (i.e. transaction) into subtransaction that can be executed by a single database at a single site. There are several reasons why this is done as opposed to splitting by sites and

letting the MDBS layers at the participating sites split the subtransactions further by databases. First, global transaction managers usually have enough difficulties in scheduling transactions across multiple sites. It may become infeasible for them to get even more involved with transaction scheduling across multiple DBMSs at one site. This would mean that a global transaction manager would send a global subtransaction to another global transaction manager at another site and expect it to coordinate the execution of the global subtransaction. In fact, if the global transaction manager needs to further decompose the subtransaction then the resulting subsubtransactions may lead to a violation of global serializability [49].

To understand some of the difficulties encountered in multidatabase transaction management, the following example is given [6].

Example 1: Consider a multidatabase environment consisting of two sites C_1 with data item a , and C_2 with data item b . Assume the following global transaction GT_1 has been submitted to the MDBS:

$$GT_1 : r_1(a)w_1(a)w_1(b)$$

Suppose that GT_1 has completed its execution at both sites and is in the *ready-to-commit* state. The MDBS now submits the commit operation to the servers at both sites. Also, suppose that site C_2 has received the commit and has successfully executed it, while site C_1 fails before the commit operation arrives. Therefore, at site C_1 the local DBMS considers GT_1 as aborted and, as a consequence, releases its local locks. Upon recovery of C_1 , a new local transaction LT_2 is submitted at the site C_1 :

$$LT_2 : r_2(a)w_2(a)$$

Transaction LT_2 can execute to completion. Following this, redoes the transaction's GT_1 write operation as a new transaction GT_3 . Transaction GT_3 is generated from the server log.

$$GT_3 : w_3(a)$$

This new transaction is then submitted and executed to completion. Hence, from the local DBMS's viewpoint, the committed projection of the local schedule generated at site C_1 is:

$$r_2(a)w_2(a)w_3(a)$$

However, GT_3 's w operation is the same as that of GT_1 as far as the MDBS is concerned. Thus, this execution results in the following nonserializable global schedule from the MDBS viewpoint, where both GT_1 and GT_2 are committed in the schedule.

$$r_1(a)r_2(a)w_2(a)w_1(a)$$

There are two main directions in research to solve such problems. One direction maintains standard serializability, but imposes various restrictions on the MDBS environment. The second approach is to redefine serializability with MDBSs in mind.

2.6.2 Restrictive Model with Standard Serializability

The work in [4-7] uses standard serializability as a correctness criterion. The claim is that if global and local transactions are serialized independently, then global database consistency is maintained given the following assumptions:

- Component DBMSs use the strict two phase locking (2PL) protocol. Hence, each local schedule is serializable. Also, any local deadlocks are detected by the DBMSs and recovered from.
- The global transaction managers of the MDBS layers at each site use the 2PC protocol.
- The situation described in example 1 cannot occur if either one of the following two conditions hold for all global transactions GT_i and all local transactions LT_i :

1. Condition 1

- $WS(GT_i) \cap RS(LT_j) = \emptyset$
- $WS(GT_i) \cap WS(LT_j) = \emptyset$

2. Condition 2

- $RS(GT_i) \cap WS(LT_j) = \emptyset$

The third assumption states that items in the component databases are distinguished into *locally updatable items* and *globally updatable items*. Items that cannot be updated locally—for example if data item a is replicated, then local transactions may not be allowed to update a —are a globally updatable items. All other items are locally updatable.

Now for condition 1 to hold, it is required that a local transaction be prohibited from reading and writing globally updatable items. In addition, global transactions are restricted to writing only globally updatable items. For condition 2 to hold, it is required that a global transaction be restricted to reading only globally updatable data items.

Note how the situation in example 1 cannot occur anymore. If a is a globally updatable item, then condition 1 states that transaction LT_2 is illegal. If a is a locally updatable item the transaction GT_1 is not allowed.

Note that even with the assumptions given above, serializability can still be violated if a failure occurs during the procession of a global commit operation. A global scheduling algorithm was proposed in [4–6] to deal with such problems.

The assumptions of this model can be easily implemented by defining read/write access privileges (for example by the GRANT in RDBMSs) to local and global users and applications on data items depending on whether they are globally or locally updatable. Recall that one of the advantages of multidatabases over homogeneous DDBMS is that they can be setup with minimal disturbance to existing applications and processes. Restricting local transaction from updating global items, specially in the case of replicas, is a reasonable demand with the justification being to maintain consistency between replicas. However, restricting global users from updating local items may undermine the usefulness of the MDBS environment.

2.6.3 Multidatabase Serializability

Many researchers argue that traditional serializability is too restrictive and hence unsuitable for multidatabases. Multidatabase serializability is proposed in [2] as a correctness criterion for multidatabase transaction management.

Intuitively, multidatabase serializability states that all local DBMSs should guarantee local serializability of their local schedules. Moreover, if an operation of a global transaction precedes an operation of another global transaction at one site (one local history), then all operations of the first global transaction should precede any operation of the second transaction in all of the local histories.

A closely related work to multidatabase serializability is *quasi-serializability* presented in [16]. In fact, the class of schedules that are multidatabase serializable happens to be the same as the set of schedules that are quasi-serializable. [16] suggests that global transactions may have to be executed serially. However, the scheduling algorithms for multidatabase serializability presented in [2] allow for some degree of interleaving between global transactions providing a higher level of concurrency.

2.7 Case Studies

Work on multidatabases dates back to the early eighties. The earlier work started at the U.S Department of Defence (DoD) with the Mermaid project [8], and at INRIA with the Sirius-Delta project [22]. Up until 1987, the work on multidatabases has only been theoretical with a few research prototypes. It was still unclear whether an operational MDBS could actually be built on a commercial level. Then, in 1987, the first commercial systems to offer multidatabase system capabilities were introduced. These included Sybase [43, 66], Empress V2 [43], Ingres/STAR [54] and later SQL*Star of Oracle [43].

In this section, we present a brief overview of some of the typical prototype and commercial multidatabase systems that pertain to the three types of Federated DBSs. It is important to note that none of the commercial or prototype MDBSs

have global update capabilities. Finally, most of these systems involve an effort in the order of 20 to 40 staff/years.

2.7.1 Loosely Coupled MDBSs

One of the better known loosely coupled MDBSs is the prototype system MRDSM (Multics Relational Data Store Multidatabase) [38–40] developed at INRIA. MRDSM is used for interoperating MRDS (Multics Relational Data Store) DBMSs as well as other Videotex data repositories. It uses the relational model as its federated data model and defines the multidatabase language MDSL (Multiple Data Store Language) which is an extension of the DSL language of MRDS database systems. MDSL has similar features as its SQL counterpart MSQL presented in section 2.4. In fact, most of the work on multidatabase languages was done in the MRDSM project.

A very popular loosely coupled MDBS is Sybase. Sybase is, actually, a commercial system due to Sybase, Inc. It is available on Sun, VAX and Pyramid machines. Sybase is a high performance relational MDBS. In addition to its own SQL Engines, Sybase provides gateways to heterogeneous DBMSs. Although, Sybase defines a client/server architecture built on top of RPC, it does not, currently, support distributed queries, only remote nonglobal queries.

Sybase defines a multidatabase language Transac-SQL. The main features of Transac-SQL is that it allows qualification of table names with database names. It also allows implicit joins where all participating (fragment) relations are at the same database. Finally, in Transac-SQL, the user has to explicitly open one database at a time through the `USE <database name>` statement.

Other MDBSs that follow the loosely coupled approach include OMNIBASE [58, 59] and CALIDA [31].

2.7.2 Tightly Coupled MDBSs

Most of the prototype and commercial MDBSs are tightly coupled. The reason is that most of these systems are extensions of homogeneous DDBMS. Also, among these systems, most have followed the multiple federation approach. This is mainly due to semantic heterogeneity problems that create additional difficulty in integrating all the component schemas into a single federated schema.

2.7.2.1 Multiple Federations MDBSs

Mermaid [65] is not a true database management system, but rather a front-end system that locates and integrates data that are maintained by component DBMSs. It is currently being developed as a commercial product by Data Integration Inc. (previously Unisys).

Mermaid supports federated views. It is a relational MDBS that uses SQL as its multidatabase language. It also defines a query-by-form language called ARIEL. Mermaid defines the Distributed Intermediate Language (DIL) as its canonical language which is used when evaluating query access plans. In fact, Mermaid provides some of the most prominent global query optimization capabilities among multidatabase system. In particular, it supports two types of optimization algorithms, the global reduction semi-join algorithm and the replicate join algorithm [8].

Mermaid defines a client/server architecture built on top of TCP/IP RPC. It supports gateways to Ingres, Oracle, IDM and even database machines.

Perhaps one of the most popular commercial tightly coupled MDBS with multiple federations is Ingres/STAR. Other systems include Xerox's MULTIBASE [61] which defines a semantic data model DAPLEX, UCLA's HD-DBMS [10], Amoco's ADDS [61, 66], CRAI's (Italy) DQS/Multistar [43] and the University of Maryland's ADMS±. ADMS± has the unique feature of database caching. That is, temporary results are not directly discarded, but are kept for future references to the same data.

2.7.2.2 Single Federation MDBSs

DDTS (Distributed Database Testbed System) [18] is a testbed system developed at Honeywell Corporate System Development Division. DDTS is one of the very few MDBSs to adopt the single federation approach.

DDTS uses the ECR as its federated data model. It defines the multidatabase language GORDAS. It also supports federated views. Currently, DDTS provides access to the relational RAM and the CODASYL IDS2 DBMSs.

2.7.3 Other MDBSs

Many developers of object-oriented DBMSs (OODBMSs) have realized that in order to be more competitive in a market dominated by relational and other DBMSs, their OODBMSs should provide gateways to these systems. For instance, the GemStone OODBMS follows this approach by providing gateway support to Ingres, Oracle and Sybase. Hence, many OODBMSs are moving towards supporting (distributed or non-distributed) multidatabase functionalities.

The Pegasus project is a prototype MDBS that follows the object oriented approach [43]. It uses Object SQL of the Iris system extended to support multidatabase functionalities. The project is being developed with the aim of supporting file systems and multimedia applications in addition to the object-oriented, relational, CODASYL and hierarchical DBMSs. The first phase of the Pegasus project is to support the Allbase RDBMS, the Iris OODBMS, the TurboImage hierarchical DBMS, and the KSAM file system.

Chapter 3

The MultiDataBase Integrator

Chapter 2 introduced some of the main issues in multidatabase interoperability. The work presented in this thesis addresses primarily the MultiDataBase Integrator—a loosely coupled multidatabase backbone system. The MDBI project has three objectives. First, it provides interoperability between DBMSs and/or GISs. The MDBI defines a multidatabase language called MDBI*SQL to achieve this. Second, since query optimization and transaction management protocols are still open areas of research, the MDBI can be used as a testbed for such protocols. Finally, although the MDBI is not a full MDBS with full functionalities¹ it can be used as an *integrator* to build an MDBS with its particular query processing and transaction management schemes.

Many of the prototype and commercial MDBSs impose various restrictions on the multidatabase environment, or require rewriting of component DBMS software. The proposed MDBI makes the least restrictions possible on the multidatabase environment and requires no rewriting of component DBMS/GIS software. MDBI, hence, supports the three dimensions of distribution, autonomy and heterogeneity. It is based on a bottom-up design process that builds on what is available in terms of data model, and off-the-shelf DBMS and GIS products. While a top-down design is more suitable for centralized and distributed DBMSs, we choose the bottom-up approach for the MDBI because it follows more the concept behind an

¹Note that none of the commercial or even prototype MDBSs provides full multidatabase functionalities.

MDBS environment. It is true, however, that often we go back to the top-down approach especially in designing the system DD/D and auxiliary schemas, and in other (multi)database management facilities.

This chapter presents the MDBI proposal. The next section gives an overview of the MDBI, its loosely coupled model, its client/server remote query execution model and its logical topology model. Section 3.2 describes the proposed MDBI*SQL language. Finally, section 3.3 discusses the proposal for GIS interoperability using MDBI*SQL. Details of the architectural design as well as the current implementation status are left to the next chapter. Finally, the current prototype and analytical performance model are presented in chapter 5.

3.1 MDBI Overview and System Layout

The MDBI is a loosely coupled MDBS. The system is subdivided into three components: the *Virtual DataBase* (VDB), the *Remote DBMS Access Interface* and the *Local DBMS Gateway(s)*. The system layout is shown in figure 3.1.

VDB is the multidatabase management software layer. It handles the various functionalities of the MDBI. This includes defining a multidatabase language and a query parser; and maintaining DD/D and auxiliary schemas. VDB uses (one of) the local DBMSs at its site to store and manage its data. Since there is no widely accepted transaction management model for multidatabases, the VDB does not implement a transaction manager. In fact, only local and remote transactions are allowed in the MDBI environment. This way, the transaction management tasks can be safely delegated to the local DBMSs.

At the lower level, the Remote DBMS Access Interface is composed of three layers: the Data Structure Interface (DSI), the Remote Network Access Manager (RNAM) and the Network Interconnection layer (NI). This network communication layer uses RPC/XDR as its base. This has two main advantages. First, the low-level details of the network communication are handled by RPC/XDR [64, 69]. The MDBI software can use RPC routines without having to worry about low level

Figure 3.1 Architectural Layout of the MDBI

socket-based interprocess communication (IPC), for example. Second, RPC/XDR provides support for hardware and operating system heterogeneity as well as network and transport protocol heterogeneity. Since the MDBI's communication interface is built on top of RPC/XDR, the network distribution and heterogeneity support of RPC are directly inherited by the MDBI. RPC is described in detail in section 4.1.

MDBI also supports full autonomy of the component DBMSs. The MDBI does not require rewriting of DBMS software to be able to interface to it. Interfacing to a LDBMS is done through a gateway layer. The gateway layer is a set of library routines used to access local DBMS (LDBMS) data. The gateway routines are not restricted by what the vendors provide in terms of gateway support. Instead they

use whatever facilities the LDBMS provides to its users and applications. For example, the Ingres and Oracle RDBMSs provide embedded C facilities with Dynamic SQL. GISs typically provide translators from between their native format and some interface ASCII file format.

The gateways also provide support for data model and DBMS heterogeneity. Our proposed MDBI is not an extension of any DBMS product, nor is it based on any particular DBMS product or even data model. Hence, the underlying LDBMS is completely transparent. Therefore, VDB and the Data Structure Interface, which use the LDBMS, both share the gateway layer. And they can only access the LDBMS through this gateway.

3.1.1 MDBI's Loosely Coupled Model

The loosely coupled model of the MDBI is different from the one defined in section 2.2.1. In our case, the MDBI defines a four level extension to ANSI/SPARC as shown in figure 3.2.

Unlike other loosely coupled models, the MDBI defines a single global federation. All participants should register their data in this federation. However, in conformance with the loosely coupled approach, no schema integration is performed. Rather, all databases in the federation are viewed as individual entities. It is up to the users to resolve semantic differences (or similarities) between these databases according to their requirements and needs.

The MDBI's loosely coupled model also defines a federation DBA. While the local DBAs are responsible for making all or part of their local schemas available to the federation in federation format, the federation DBAs are responsible for including these export schemas into the federation by importation. Recall from section 2.2 that exportation and importation do not physically move or migrate the data.

Finally, the MDBI uses the relational model at the federation level. Therefore, the relational model, including the concept of databases (e.g. databases of Ingres), is the conceptual model of the MDBI. The export schemas in figure 3.2 also include the component schemas as defined in section 2.2. However, in our case, exportation

Figure 3.2 The loosely coupled model for MDBI

is performed before translation. That is, only exported data need to be (logically) translated into the federation model.

Notice that our loosely coupled model is also similar to the tightly coupled model. The advantage of this is that tightly coupled functionalities can become easier to build in to the system. This approach, hence, provides better support for eventual update capabilities. Therefore, we term also our model as *lightly coupled*.

It is desirable that federation views (i.e external schemas) be defined on top of the federation schema, but such a feature is not available in the current MDBI proposal.

3.1.2 Client/Server Model for Remote Queries

The MDBI supports global query capabilities. Global queries are interdatabase queries that span multiple sites. MDBI implements a query processor (see section 4.2.2) that splits and fragments a global query into global subqueries as defined in section 2.5. These global subqueries are now remote queries expressed on one database at one site. MDBI defines a client/server model to transmit remote subqueries into local sites and get the results back to the initiator site. Such queries are executed in two phases: the *client phase* and the *server phase*. At the client phase, the initiator site acts as a client and transmits the remote queries to the participator sites which act as servers. At the server phase, the participators act as clients and send back the results of the queries to the initiator site, which now becomes the server.

In this model, suppose site A wants to execute a remote query at site B, but requires the results back at A. This is done by packaging the query in a network block. Sending it to site B. Unpacking the query and executing it against the database of site B. Packaging the result into network block(s) format. Sending the results back to site A. And finally, recreating the results at site A. Although this is intuitively simple, the model constitutes a powerful base for the MDBI. The MDBI easily extends this model to support multiple DBMSs, multiple sites and even multiple concurrent queries.

The model switches between client and server to promote concurrency and hence response time and throughput. The client of the initiator terminates after it sends its request to the server, so the initiator can proceed with other operations such as sending another subquery to one of the other participators. On the other hand, the server of the participant forks a process to handle the DBMS access. This allows the server to service other incoming requests since DBMS and specially GIS access can take a substantial amount of time. Further details on the system's execution protocol are given in section 5.2.1.

3.1.3 Logical Topology of the MDBI

The MDBI establishes what we refer to as *two-way fairness* between all the databases in a federation. Typically, commercial MDBSs, including Ingres/STAR, Oracle's SQL*Star and Sybase have logical star topologies. For example, if Ingres/STAR is running at a certain site, then it can access data from any site running Ingres/NET, but the converse may not always be true. Hence, users of the Ingres/STAR site can access data of participators, but participating sites cannot access data from each others nor can they access federation data. This is more accentuated when participators are non-Ingres DBMSs, including SQL/DS, DB2 and RDB.

In an MDBI environment, any participator can access any other participating site. The DBMS model or type does not restrict its access. Even component GIS databases in an MDBI environment can still access data from any participator. Hence, each site can be thought of as being a star. Put together, all the sites will, therefore, represent a complete logical topology. This is what is meant by two-way fairness. It is not only relevant to data access capabilities, but also to the performance model of the MDBI as we will see in section 5.2.

Note that because the Remote Procedure Calls and Internet Protocols are used by the MDBI, the physical topology of the network is abstracted from the MDBI environment. Each site "thinks" that it can directly communicate with any other site.

3.2 The MDBI*SQL Multidatabase Language

MDBI*SQL supports update as well as retrieval queries. However, since transaction management in multidatabases is still an open area of research and because the MDBI is a loosely coupled MDBS, global (distributed) update queries are not permitted. Only, local and remote update queries are supported, where remote queries are queries expressed on a single database at a single, possibly remote, site. Hence, this section focuses on retrieval queries only.

Not only are the proposed multidatabase features in MDBI*SQL useful in a multiple DBMS environment, but they have also been designed with a multiple GIS environment in mind. This will become more apparent in section 3.3 below.

3.2.1 SELECT queries

The SELECT statement of MDBI*SQL is based on its counterpart in Open SQL [56] to maximize its compatibility with commercial RDBMSs and hence its usefulness. The general syntax of a SELECT query in MDBI*SQL is as follows:

```
subselect {UNION [ALL] (subselect)}  
[ORDER BY result_column [ASC | DESC]  
        {, result_column [ASC | DESC]}]
```

The syntax of the **subselect** clause is as follows:

```
SELECT [ALL | DISTINCT] expression {, expression}  
[RENAME AS table_expression [NOWAIT]]  
FROM   table_expression [alias] {, table_expression [alias]}  
[WHERE search_condition]  
[GROUP BY column_expression {, column_expression}]  
[HAVING search_condition]
```

Words shown in lower case are nonterminal symbols². Words shown in capital letters represent key words (terminal symbols). Optional clauses are enclosed in brackets ([]). Clauses enclosed in braces ({ }) are optional and can be repeated zero or more times. Finally, words separated by a bar (|) represent lists from which only one element should be chosen.

An **expression** may be any legal expression in a SELECT clause in standard SQL [55]. Set functions including COUNT, MAX, MIN, SUM and AVG, in addition to arithmetic operations, are all supported. The **search_condition** is any

²Using compiler terminology.

legal standard SQL predicate expression. These expressions may include comparison predicates (<, <=, =, <> etc), LIKE predicates, BETWEEN predicates, IN predicates, ANY or ALL predicates, EXISTS predicates and IS NULL predicates. Subqueries in search conditions are also supported.

The RENAME clause is proposed for MDBI*SQL SELECT queries. The RENAME clause is allowed only in the first **subselect** clause. The purpose of the RENAME clause is to save the result of a SELECT query as an object (say a table) with the given table name at the local site. This is equivalent to the CREATE AS SELECT query in centralized DBMSs. However, MDBI*SQL does not support global CREATE queries and so this syntax is required when a user is interested in creating a table from data spanning multiple databases. The optional NOWAIT tells the system not to wait for the results of the SELECT RENAME query, and proceed with other user requests.

A **table_expression** is a qualified table name. Multidatabases typically qualify table names by the names of databases. Some even allow qualification by site name. For such systems, **table_expression** may have the following syntax:

[database].tablename

These facilities are needed because of table name conflicts and even database name conflicts between multiple schemas. The following extension to traditional qualifiers is proposed for MDBI*SQL:

[dbms@][sitename.][database.]tablename

All of the qualifiers for table names are optional. The use of any qualifier does not necessarily imply the use of any other. In this case, **tablename** may actually be a table name in an RDBMS, a feature or a theme of a map in a GIS. Eventually, it may even be a file name, a CODASYL set name, an object category name etc.

At the conceptual level of the MDBI, **database** is a database name. In the relational model, **database** actually refers to the name of a database (e.g for Ingres) or an alias for a user workspace (e.g for Oracle). For file systems **database** is

an alias representing a directory name. Finally, **database** actually refers to an alias representing a map name and its directory. For GISs, this organization closely follows the Delta-X [17] model. In Delta-X, Relational tables represent map features, themes and attributed, and databases represent maps as discussed in section 3.3.

The name of a site as it is known in its network domain is represented in **sitename**. Note that **sitename** is redundant in the RENAME clause because it must always refer to the local site.

If multiple DBMSs are running on the same site, then **database at sitename** may no longer be unique. The **dbms** name can be used to solve such ambiguities. More importantly, this feature gives the user the ability to access a specific database system at a given site. For example, ARC/INFO provides two subsystems ARC and INFO. If a user only wants vector data from a map managed by ARC/INFO, then he can specify **arc**. On the other hand, if the user is interested in accessing only attribute data, then he can specify **info**. Finally, if the user wants both vector and attribute data, he or she can specify **arcinfo**. The same is true for CARED, CARMAN and CARIS respectively as well as different versions of the same product etc. In general, this feature is useful when more than one "DBMS" resides at a certain site. This feature is especially useful for the RENAME clause. It allows the user to decide on the DBMS or GIS under which the results of his or her SELECT query will be stored.

Columns can also be qualified. Column qualifiers are either aliases or qualifiers similar to table qualifiers. Hence, column expressions have the following syntax:

[dbms0][sitename.][database.][tablename.]columnname

or

[alias.]columnname

When the alias is used as a qualifier, there is no need to use any other qualifier. This is similar to the case of standard SQL where an alias uniquely identifies a table name in the FROM clause.

Qualifiers for column names are needed whenever there is a conflict between two columns of two (not necessarily) distinct tables. In the case of such conflicts, any

one or more of the qualifiers can be used as long as the conflict gets resolved. Note that the qualifiers can be used for the columns even if they have not been used by the corresponding tables. Such qualifiers are termed as *implied* qualifiers.

Assuming that two tables **Cars** and **Motorcycles** have a column **brandname**, where **Cars** is stored at a site **germany** and **Motorcycles** is stored under database **racing**. Then, assuming that both tables have been defined in one or more multi-databases, the following query is legal:

```
SELECT racing.brandname, germany.brandname
FROM   Motorcycles, Cars
WHERE  ...
```

This is because, as we will see in section 4.2.2, table names are fully qualified, before columns are matched with tables.

The advantage of this feature is that it allows the user more flexibility in choosing column qualifiers. In particular, it allows the users to use any qualifiers they want thus making their queries shorter and more compact as in the following query:

```
SELECT registration.name
FROM   ingres@socrates.registration.student,
       ingres@socrates.financialaid.student
WHERE  ...
```

Note that the same syntax applies to column references in the WHERE, HAVING, etc clauses.

The disadvantage of this feature is that any column that references a table that is multiply defined in the FROM clause *must* be qualified by an alias. Because all table names are fully qualified before being matched with their columns, none of the qualifiers (except an alias) will resolve the ambiguity. The same is true in standard SQL.

Assuming that the two references in the FROM clause are those of the same **majors** table, the following query is ambiguous:

```
SELECT registration.faculty
FROM   registration.majors, majors
...
```

The fact that one reference to table **majors** has been qualified with the database name, or any other qualifier(s), does not distinguish it from the other reference to that same table.

Other queries including CREATE TABLE, INSERT, UPDATE, DELETE and DROP TABLE all have the same syntax as the corresponding commands in standard SQL, except that **table.expressions** and **column.expressions** are allowed. Reminder that these transaction queries can only be local or remote, but not global, queries. A DESCRIBE query is also proposed for MDBI*SQL. It is similar to the DESCRIBE command in SQL*Plus of Oracle. DESCRIBE returns the structure of its argument **table.expression**.

3.2.2 COPYBD queries

In MDBI, maps are represented as databases at the conceptual level in conformance with the Delta-X format. The COPYDB command is proposed for copying a (subset of a) database between two database systems possibly running at two different sites. COPYDB has the following syntax:

```
COPYDB database_expression{,tablename}
      INTO  database_expression
```

Similarly to its table counterpart, **database.expression** has the following syntax:

```
[dbms0][sitename.]database
```

Note that **sitename** in the INTO clause is currently redundant as it must always refer to the local site. The COPYDB command creates the target **database.expression**. Hence, a database with this name should not have already been created under the corresponding **dbms** at the local site. Finally, when one or more **tablename**s are specified in the source database, only these tables are copied into the target database.

3.2.3 EXPORT queries

Local DBAs can make all or part of their schemas available to the federation using the EXPORT commands. Recall that an export schema contains only the definitions of the exported data. The data is *not* physically migrated.

Local DBAs can export a whole database using the following command:

```
EXPORT DATABASE database_expression_2 {, database_expression_2}
      AS database {, database}
```

Where `database_expression_2` has the following syntax:

```
[dbms@][IP_address:][sitename.]database_2
```

The optional syntax is only required if one or more of the `dbms`, `IP_address` and `sitename` have not already been exported. The optional syntax is also needed in case of conflicts. The Internet Protocol (IP) address is used since MDBI is built on top of RPC TPC/IP for internets.

GISs use the concept of a map in a directory and some DBMSs use the concept of a user workspace instead of a `database`. Therefore, the syntax of `database_2` is defined as follows.

```
identifier{/identifier}
```

For example, to export a Caris map `acme.acres` in directory `/home/users/mrx` stored at site `gianni` with IP address `9.9.9.9`, the following commands are issued.

```
EXPORT DATABASE caris@9.9.9.9:gianni./home/users/mrx/acme_acres
      AS acme_acres
```

Also, to export the workspace of user `spock/xxx`, the following commands can be issued.

```
EXPORT DATABASE oracle@9.9.9.9:gianni.spock/xxx
      AS spocks
```

Of course, a relational database **registration**, such as in the case of Ingres can be exported simply as:

```
EXPORT DATABASE ingres09.9.9.9:gianni.registration
      AS registration
```

The target name always represents a true database name. It is called a *database alias name* when it actually refers to a non-database as in the case of GISs and file systems. Note that in all cases, it is the responsibility of the local DBA to determine the type of the “database”. Once that “database” has been exported, it becomes a conceptual **database**. Therefore, the federation DBA and all federation users can treat it as a relational database. The actual details are, hence, hidden from the federation.

Often, the local DBA want to export only a subset of a database. This is done using the following syntax:

```
EXPORT TABLE database_expression_2.tablename{,tablename}
      AS database_expression.tablename{,tablename}
```

This syntax has the effect of also exporting the definitions of the “databases”, site names etc in addition to those of **tablename**. For example, to export tables **students** and **courses** of Spock’s workspace, the following commands can be issued.

```
EXPORT TABLE oracle09.9.9.9:gianni.spock/xxx.students,courses
      AS spocks.students,courses
```

Note that in both cases of the EXPORT syntax, the number of source and target objects to be exported should always match.

3.2.4 REGISTER queries

EXPORTing data does not make it readily accessible from the federation. The EXPORT queries have the effect of making data available for the federation as well as expressing this data in the MDBI's conceptual model. Referring back to section 3.1.1, EXPORT defines the export schemas.

To make the data become part of the federation, the federation DBA has to import the exported data. This has the effect of *registering* the data in the federation. This is achieved using the REGISTER queries. Hence, in section 3.1.1, REGISTER defines the federated schema.

To register exported databases, the following syntax is used:

```
REGISTER DATABASE database_expression {,database_expression}
```

and to register exported tables, the following syntax is used:

```
REGISTER TABLE database_expression.tablename{,tablename}
```

The qualifiers of **database** are only needed in case of a conflict between exported database names.

Once a table or a database is exported and registered, it becomes accessible to federation users. As in the case of EXPORT, REGISTER does not create nor migrate any data. The effect of REGISTER is to define the appropriate entries, also called *links*, in the DD/D and Auxiliary Schemas.

It is important to note that while the data has been made available to the federation, the users must still have authorization (say GRANT) on the data to actually be able to access it.

3.3 GIS Interoperability with MDBI*SQL

This work proposes MDBI for multidatabase interoperability between DBMSs and/or GISs. This proposal is unique to the MDBI project. However, there is another proposal for GIS interoperability only.

3.3.1 Previous Work

The author is not aware of any proposal (other than MDBI) for a GIS multidatabase except VPF View. This is an ongoing project at DoD, with the cooperation of the Environmental Systems Research Institute (ERSI). The project is still in its Beta phase.

VPF View has a Multiple DB component built on the DCW database system. This system defines a query language based on SQL. DCW data is stored in CD-ROM databases made by ESRI. The format of the data follows the DIGEST Vector Relation Format (VRF). That is, the DCW is a relation data repository. Translators are also being built to allow data exchange with the ARC/INFO and the Spans GIS products.

The main limitation of the VPF View system is that it only allows interoperability between GIS data stored in VRF under DCW. It does not allow interoperability with off-the-shelf DBMSs. Another limitation is that VPF View can only make direct access to its own data repositories; i.e., DCW.

Among the current GIS products, Vision 2.0 by GeoVision seems to hold the most promise in terms of providing multidatabase capabilities. Unlike other GISs, Vision 2.0 uses a Oracle to store its vector as well as its attribute data. Since Oracle provides multidatabase capabilities in the SQL*Star subsystem, these can potentially be inherited by Vision 2.0. However, Vision 2.0 is limited by what oracle provides in terms of interoperability with DBMSs. GIS interoperability has not been proposed for this system.

Finally, there have been many proposals for directory services for GIS databases. One of the better known directory systems is GC/Net. Such systems only provide participators with directory information for data sets that users may be interested in. Although loading these data sets can be done over the network, it is still a manual process.

3.3.2 Interoperability of GISs and/or DBMSs

Unlike VPF View, MDBI proposes interoperability between DBMSs and/or GISs. It does not define nor use any particular data repository. Rather, MDBI proposes that data, including the MDBI's DD/D and Auxiliary Schema, can reside in any participating off-the-shelf DBMS or GIS product.

In the previous section, we have extended MDBI*SQL to support multidatabases functionalities (distribution, autonomy and heterogeneity). Because MDBI is a loosely coupled MDBS, MDBI*SQL becomes the core of the system including the support for GIS databases.

Having a multidatabase language can also be used as a basis for building tight coupled functionality on top of the MDBI. For instance, [47] proposes the Spatial Browser, which provides functionalities similar to those of directory systems. MDBI can, hence, use the Spatial Browser to mimic the functionality of an integrated schema in the case of heterogeneous GIS databases.

The Delta-X and the Spatial Browser have been proposed as two complementary systems to provide support for the type of problems faced by GIS application developers [48]. The Spatial Browser allows users to locate the data they are looking for, and the Delta-X system, with its translators, allows the users to load the data off the network and in many of the off-the-shelf GIS product formats.

The Delta-X project proposes the Delta-X format [46]. This new vector and attribute data format is based on relational DBMS technology. It defines a schema that captures most, if not all, the topological relationships between map features. The basic components of a map tile in Delta-X format are **NODES**, **LINES** and **POLYGONS** and are called *features*. These features are organized into **LAYERS**, **OVERLAYS**, **TILES** etc that are called *themes*. The Delta-X format supports versions, on **LAYERS** and even on the individual features of a map tile. It also supports multiple topological levels including no topology, node topology, arc topology and polygon topology on the same map tile or even on the same layer. Delta-X maps are therefore represented as databases and their features and themes are represented by tables in relational DBMSs. An added advantage of using the Delta-X format is that the facilities of

Figure 3.3 Distributed GIS interoperability with MDBI*SQL

the RDBMSs can be used to build customized and general purpose applications to complement the analytical tools that GISs may (or may not) provide. Finally, the Delta-X project defines translators between the Delta-X format, and the formats of GIS products and interchange formats such as CCOGIF, DLG and SIF.

Although GISs are an application of database systems, they lack many of the features of traditional DBMSs; not the least of which are high level nonprocedural ad-hoc query languages. This brings us back to the problem of pre-database integration discussed in the beginning of Section 2.3. There are two ways of solving this problem. The first way is to add an extra layer in the Gateways to allow SQL support for GISs. Alternatively, GIS data can be migrated to DBMSs; that is, the Delta-X format.

The MDBI adopts both approaches. The MDBI readily supports RDBMS and hence the Delta-X format. In addition, using the Delta-X translators, the MDBI also provides SQL support for GIS. Hence, the MDBI, with MDBI*SQL at its core, completes the "puzzle". By providing a flexible ad-hoc high level nonprocedural language, it allows simultaneous and seamless distributed access to multiple maps from multiple GIS and DBMS repositories. This is depicted in figure 3.3.

While most GISs still store their vector data in files, many provide an interface to a relational database, typically Oracle or Ingres, to store their attribute data. Therefore, the Local DBMS Gateways to relational databases can be readily used to access attribute data of GISs.

GIS vector data is usually stored in binary files and, hence, are not suitable for network communications in a heterogeneous environment. In addition, the vendors often do not provide a detailed description of the format in which the data is stored. The Local DBMS Gateway layer should provide special interfaces to GIS products that maintain a high degree of autonomy for the GIS. These interfaces use facilities provided by the GIS product to upload and download geodata into ASCII files. By incorporating the Delta-X translators into the Local DBMS Gateways, the MDBI currently provide access to ARC/INFO and Caris. Therefore, MDBI uses the Delta-X format as the canonical data format between heterogeneous GIS databases.

The Gateway to ARC/INFO uses the GENERATE command to load vector data from an ASCII file. Such a file corresponds to an ARC/INFO coverage and contain the x-y coordinates of features in that coverage. The Delta-X translators are first used to load the ASCII file(s) from the Delta-X format. To download an ARC/INFO coverage, the Local DBMS Gateway invokes the UNGENERATE command. This command creates an ASCII file corresponding to a coverage. The file contains the x-y coordinates for the features in that coverage. Again, the Delta-X translators are used to load the ASCII file into Delta-X.

The facilities and commands provided by Caris are quite different. To load data into Caris, or to digitize features in Caris to use Caris terminology, the CARED command is used in batch mode on the ASCII file. On the other hand, to download a map sheet from Caris, our interface uses the REFOCCSM or the REFODLG commands. These commands create corresponding CCOGIF or DLG files. Delta X translators can also be used to upload/download data between the interchange file formats and the relational Delta-X format.

To see how MDBI*SQL can now provide distributed interoperability of heterogeneous GIS products, consider a typical environment for GIS application develop

ment.

Example 1. A certain Mr. X is faced with the following problem. He has been assigned the task of performing an analysis study on a geographical area. Unfortunately, the map data for that region is not available on the GIS of his local workstation. However, the various layers of this map are stored as separate maps on different GISs, on different workstations in his division, and in other divisions or agencies with a different network domain(s).

Due to the lack of a multidatabase GIS environment, the only option available to Mr. X is to download the required maps into an interchange format (such as DLG) since these maps may be stored under different GIS platforms. Next, he has to remotely copy all the resulting files that are in his network domain. However, those that are in other domains must be copied using FTP. Finally, he has to somehow combine all these maps into layers of the same map using facilities that may be available on his GIS product. Note, medium sized maps are in the order of several Mbytes.

Using the MDBI approach, Mr. X can use the COPYDB queries to copy the whole or part of maps, or the SELECT queries to copy or retrieve features and themes from those maps. For example, assume that Mr. X is interested in copying the map `toon_town` into the Caris of his local work station. Also, assume that `toon_town` has already been defined in the federation and is currently residing under ARC/INFO at site `giorgio`. Using MDBI*SQL, Mr. X can simply issue the following commands. Recall that a map is represented as a database in Delta-X and hence at the MDBI conceptual level.

```
COPYDB arcinfo@giorgio.toon_town  
      INTO caris@toon_town
```

Note that the qualifiers would not be needed if there are no conflicts.

If Mr. X is only interested in some features, say `LINES`, (or themes, say `LAYERS`) of `toon_town`, then the following query can be issued. Recall that features are represented as tables.

```
SELECT *  
RENAME AS caris@toon_lines  
FROM   arcinfo@giorgio.toon_town.lines
```

Of course, column attributes of features (i.e tables) can be used int the SELECT, WHERE etc clauses as they would be normally used for any table.

In short, using the multidatabase capabilities of MDBI coupled with the Delta X project, GIS application developers can use MDBI*SQL as a flexible tool for distributed GIS interoperability.

Chapter 4

Architecture of the MDBI

The previous chapter introduced an extensible and flexible system that acts as the backbone of an MDBS. The MDBI is proposed for interoperating DBMSs as well as heterogeneous GIS databases. This chapter describes the architecture and the current implementation status of the MDBI.

The *Remote Procedure Calls* (RPC) facility is the backbone of the MDBI's network communication interface and indeed of the whole MDBI. In addition to network, hardware and operating system heterogeneity support, the MDBI also inherits the client/server architecture of the Remote Procedure Calls model. Recall from section 3.1 how a remote (sub)query is executed in two phases: the client phase and the server phase. There is a client and a server component at each site where the MDBI is running. In terms of figure 3.1, the system can be thought as being split vertically into a client component and a server component, which are not necessarily distinct. In fact, the client and server are only distinct at the Remote DBMS Access Interface, but they share the same VDB and Local DBMS Gateway(s). The latter two can, hence, operate in client or server mode accordingly. This is clearly different from what other MDBS's typically provide. In these systems, the gateways to "foreign" DBMSs can only operate in server mode. So, they can only receive requests for data. The converse is not true. Consider again the Ingres/STAR and SQL/DS example of section 3.1.3. Since SQL/DS applications have not been discarded by the MDBS environment, it is very likely that these applications may want to access,

in addition to their old data, the new federation data.

We start by giving a brief overview of RPC and justification for their usage. The following sections describe each of the components of the MDBI.

4.1 Remote Procedure Calls Model

The RPC facility [35,64,69] is a library of procedures that provides interprocess communication designed specifically for network applications. RPC implements a logical *client/server* communication model. The *client* process is the caller. The *server* process is the receiver. In this protocol, the server executes a procedure call on behalf of the client, as if the client had executed that procedure in its own address space. Because of this separation between client and server, they need not execute on the same machine. For this reason, anything built on top of RPC inherently becomes a network application. Referring back to section 2.1.3, the RPC model implements the four upper layers of the ISO/OSI architecture.

The RPC model is similar to the traditional procedure calls. There is one thread of control that logically winds through the two processes, the client and the server. The client is the initiator of the network communication. It sends a call message to the server process. The message contains among other things the procedure's parameters. After sending the call, the client waits, i.e., is blocked. On the other side, the server remains dormant until it receives the call. The server process, then, extracts the parameters from the message and invokes a dispatch routine. The dispatch routine performs whatever service is requested, computes the result, and sends back a reply message. After that, the server will become dormant again awaiting the next call. The reply message is received by the client and contains, among other things, the result of the procedure. The client can now resume its normal execution. Hence, in this protocol, only one of the two processes is active at any given time.

Although this is the more conventional RPC communications model, RPC does not make any restrictions. RPC calls can be made asynchronously. For example,

a client may do useful work while waiting for a reply. A server can create tasks to process a call making it free to receive other incoming requests.

A server is a piece of software where network services are implemented. A network service is a collection of one or more remote programs. A remote program implements one or more remote procedures. A server may support more than one version of a remote program in order to be forward compatible with changing protocols. Hence, RPC procedures are assigned and are uniquely identified by a program number and procedure number. In addition, each program also has a version number, so when minor changes are made to a remote service, like adding a new procedure, a new program number need not be assigned. Also, the RPC environment is inherently extensible. It supports flexible integration of new applications and network services.

RPC program numbers are also called port numbers. For user applications, allowable values for port numbers are between 0x20000000 and 0x3fffffff. A server can be invoked either by initiating it directly or through the service daemon `inetd`. In either case, the server must register first its port number with the portmapper system service. The portmapper is exclusively used by RPC programs. It is a kind of registrar that keeps track of the correspondence between ports (logical channels) and services on a machine. It provides a standard way for a client to lookup the port number of any RPC program supported by the server.

Messages are not sent to the receiver process but rather to the port at which it waits for messages. When a client wants to invoke a server program, it uses the portmapper to lookup the port number of that program. While network services register their port numbers statically or dynamically, the portmapper has the well known dedicated port number 111. The client contacts the server by first calling its portmapper. If the server is registered, its port number is returned to the client in an RPC reply message. The client can now proceed to send its RPC call message to the server's port.

RPC assumes the existence of the eXternal Data Representation library (XDR), which is a standard for encoding data in a machine-independent way. XDR is a spec-

ification for the portable data representation standard. It is used for transferring data between different computer architectures. The XDR library is a set of routines that convert data from the machines' particular representation to the network standard XDR format. This is called serialization and deserialization respectively. XDR also allows users to have standardized access to sockets without being concerned about the low level details of socket-based InterProcess Communication (IPC).

XDR allows programmers to describe arbitrary data structures in a machine independent way. This is the backbone of RPC in that data for remote procedure calls is transmitted using it as a standard. RPC uses XDR to ensure that network data is represented the same on different computers, operating systems and computer languages. For example, XDR has been used to communicate data between such diverse machines as Sun, VAX, IBM-PC and Cray. Any program running on any machine can use XDR to create portable data by translating its local representation to the XDR standard. Also, any program running on any machine can read portable data by translating the XDR standard representation to its local equivalent.

Therefore, RPC client and server programs should define XDR routines, or filters, that are used to translate data between XDR and local formats. XDR defines a data description language, as opposed to a programming language. Some of the syntax of XDR is not unlike C. RPC also defines a language which is a direct extension of XDR by introducing the program and version types. Hence, RPC programs are definitions of the data structures used for sending and receiving calls as well as the specification of the server's program, version and procedure numbers. The compiler, `rpcgen`, can be used to generate the XDR routines, and the client and server skeletons. The `rpcgen` technique is sufficient for most network applications and has the advantage of hiding the RPC/XDR programming details from the application programmers.

RPC/XDR supports a variety of transport protocols including TCP/IP and UDP/IP. While the underlying transport protocol is transparent to the applications. It should be emphasized that RPC does not attempt to provide any reliability except what the transport protocol already provides. So applications built on RPC should provide their own reliability mechanisms. For example, applications that use

an unreliable transport such as UDP/IP should implement their own retransmission and time-out policies. In addition, the servers should be made idempotent.

RPC are a high level communication paradigm. They allow network applications to be developed by special kinds of procedure calls that hide the details of the underlying network mechanisms from system programmers. Application programs use RPC to provide network services to their callers without even requiring that they be aware of the existence and functions of the underlying network. In short, RPC/XDR provide location, operating system, machine and network transparency in a client/sever framework.

4.2 The Virtual DataBase

VDB is the multi-database manager of the MDBI. This is the central component of the MDBI layer on each site. This component provides the site/database transparency of the MDBI and hence the name Virtual DataBase.

VDB is composed of two main components. The Strict VDB and the VDB Gateway. Strict VDB interacts with the user interface to receive user requests. It is also responsible for coordinating with the Remote DBMS Access Interface. Most importantly, Strict VDB defines MDBI*SQL, the multidatabase language of MDBI, by implementing the Multidatabase Query Parser described in section 4.2.2. Eventually, a transaction manager can also be implemented at this level.

The VDB Gateway is the VDB's interface with the Local DBMS Gateways. It interacts with the user interface to return the results of the user requests. VDB Gateway also implements the Local DBMS Translators presented in section 4.2.2 as well as the DD/D and Auxiliary Schema Manager discussed in section 4.2.3.

4.2.1 Interactive MDBI*SQL User Interface

Currently, the only user interface supported by MDBI is the Interactive MDBI*SQL User Interface. It is similar to Ingres's Terminal Monitor and especially to SQL*Plus of Oracle.

At the **MDBI*SQL** prompt, the user can enter pseudo-commands, or **MDBI*SQL** queries. Pseudo-commands are special commands used by the user interface to set user options as well as to get help and exit the interface. Currently, the following pseudo-commands are used:

- **/h[elp]**: Gives help information on all pseudo-commands. Eventually, this should be extended to a comprehensive help system for **MDBI*SQL** commands.
- **/e[xit]**: Quits the Interactive **MDBI*SQL** User Interface.
- **/n[et timeout] = <value>**: Sets the timeout for each network communication to **<value>** in seconds. If no timeout is set, the system uses a default timeout defined in the Auxiliary Schema.
- **/t[ask timeout] = <value>**: Sets the task level timeout. That is, the timeout for the whole **MDBI*SQL** global query. If none is set, the default setting defined in the Auxiliary schema is used.

Note that letters enclosed in square brackets (**[]**) are optional. Also, all pseudo-commands start with (**/**).

MDBI*SQL queries may span for several lines. However, all queries and pseudo-commands must be terminated by a semicolon (**;**).

4.2.2 The Multidatabase Query Processor

MDBI defines the multidatabase language **MDBI*SQL** by implementing the Multidatabase Query Processor (MQP) at the VDB component. Recall from section 2.5 that multidatabase query processing is divided into two main phases. The first phase is to split and fragment a global query into global subqueries. This phase is performed by the query processor of the initiator site. Optionally, the global subqueries are also translated into a canonical query language. The second phase is performed by the participators. In this phase, the global subqueries (possibly in

canonical form) are translated into the native language of the local DBMSs. The two phases were depicted in figure 2.7.

MQP follows that same model as that presented in section 2.5. The MQP at the initiator site is responsible for splitting, fragmenting and translating a global MDBI*SQL query into global SQL subqueries that can be executed by one database at one site. Therefore, MDBI*SQL is the *external* common query language, while standard SQL is the *internal* canonical query language, in the MDBI environment. All request to local and remote sites are sent as standard SQL queries.

The Local DBMS Translators are the only component of the MQP that are dependent on the component DBMSs at the local site. Hence, while MQP is strictly part of the VDB, the Local DBMS Translators are part of the VDB Gateway. When the native language of a local DBMS is different from SQL, the Local DBMS Translators are used to translate the global SQL subqueries into queries in the DBMS's native language. Note that similarly to the multidatabase query processing model of section 2.5 (figure 2.7), translation from canonical (SQL) to native query language is done by the MDBIs at the local sites of the participating DBMSs.

The MQP at the initiator site does most of the syntax and semantic checking. It mainly checks for the proper syntactic formulation of the global MDBI*SQL query, but cannot detect errors like type mismatches in a predicate condition. This initial step is done by the query parser of MQP. This parser generates a parse tree for a given query. The parse tree is identical to the original query except that all tables (including their qualifiers and aliases) and columns (including their qualifiers) are replaced by placeholders. All table names and their given qualifiers are stored in a table¹. The same is done for all column names and their qualifiers.

Using these tables, MQP, then, checks for alias ambiguities. These ambiguities include aliases that are multiply defined, or aliases that conflict with table (map, object etc) names.

Next, MQP fully qualifies all table names. At this step, the MQP uses the DD/D and Auxiliary Schema Manager to get the missing qualifiers. At this point, MQP

¹The term table here refers to a table in compiler terms, not a relational table.

rejects a global query for one of the following reasons. A global query is rejected if it references a table name (plus its qualifiers) that is ambiguous. That is, not enough qualifiers were given in the FROM clause and so the **table_expression** matches with more than one table definition in the DD/D. A global query is also rejected if a table and/or its qualifiers are not defined in the DD/D.

Once all the tables have been qualified, MQP proceeds to matching columns with tables. Priority is given to aliases. An alias forces columns to be matched with the corresponding tables. The other qualifiers may or may not be enough to match a column with a table. Also, some column references may not be qualified at all. In the latter two cases, the DD/D and Auxiliary Schema Manager is used to match columns with tables given the definitions in the DD/D. A query is rejected if a column reference matches with more than one table, or if it does not match with any tables. A query is also rejected if using the qualifiers of a column (including alias), the column is matched with a table that is defined in a federation, but its definition does not include that column.

In terms of section 2.5, MQP follows the simple GDM and LDIs approach. The results are merged at the originating site and the global subqueries are generated to be the smallest possible local queries.

MQP generates the *merging* query before splitting fragmenting the global query. The merging query is the query that will merge all the results, received from the participating sites, into the final result. The merging query is identical to the parse tree, but where MQP replaces the table name placeholders in the parse tree with the name of the temporary results that will eventually be received. It also assigns an alias for each "table name" and uses the aliases to qualify all column references. Note that for optimization purposes, duplicate references to the same table name, in a global query, are assigned the same temporary result name (but not the same alias).

MQP follows the simple GDM and LDIs approach. MQP does not group references to all tables into the same site/database in one global subquery. Rather, it generates a distinct global subquery for each (distinct) table reference. The MDBI

project was originally planned for remote (versus global) queries only. Due to time constraints, no comprehensive global optimizations have been implemented when the MDBI was extended to global query capabilities. Only two types of optimization are performed. First, only one global subquery is produced for each distinct reference of a table name. Even if a table is multiply referenced (including in subqueries), only one global subquery will be generated.

Second, projection optimization is performed. Global subqueries are projected on all the corresponding columns that are referenced in the global query. Thus, global subqueries have the following syntax:

```
SELECT <all references to columns of table X>
FROM   <a distinct reference to table X>
```

Note that global subqueries contain no qualifiers. Hence, this step has also the effect of translating MDBI*SQL into standard SQL. Each generated global subquery is submitted to the Remote DBMS Access Manager along with the qualifiers as *task parameters* for transmission to the remote site. The task parameter's structure is as follows.

```
struct taskparam {
    char *thishost;      /* name of local (initiator) site */
    int  taskid;         /* site unique task id number */
    char *thisipaddress; /* IP address of local site */
    char *dbms;          /* remote DBMS of subquery */
    char *dbname;        /* remote database of subquery */
    char *server;        /* IP address of remote site */
    int  timeout;        /* timeout for a communication */
    int  tasktimeout;    /* timeout for the task */
    char *resulttablename; /* table name of result */
    int  resulttaskid;   /* task id of the result */
}
```

The DBMS, database name and IP address information is assigned to **dbms**, **dbname** and **server** from the query qualifiers and/or the DD/D. The name of the result table in the RENAME clause, or the name of the temporary results are assigned to **resulttablename**. The other information is assigned from the Auxiliary Schema as described in section 4.2.3.

Note that if a transaction manager is to be implemented, it should be implemented at this level. Hence, the transaction manager will receive the global subqueries from MQP and submit them to the Remote DBMS Access Interface as specified in section 2.6.

The Remote DBMS Access Interface has the task of sending the global subqueries to the participating sites. The global subqueries are received by the Remote DBMS Access Interfaces of the MDBIs running at these sites. These subqueries are then submitted to the VDBs. In particular, if the native language of the local DBMSs is not SQL, the Local DBMS Translators of the MQP will pick up the global subqueries and translate them into the native language of the DBMSs. Currently, MDBI supports only SQL DBMSs, so this feature is only used for GISs.

An SQL SELECT query expressed on GIS map features is translated into a series of GIS commands that will allow the data of the map to be extracted. A similar discussion holds for COPYDB queries.

An SQL query on ARC/INFO is hence translated into a series of UNGENERATE commands, one for each coverage² of the map. The UNGENERATE command has the effect of downloading the vector data of a coverage into an ASCII file with a given format. In ARC/INFO, coverages are represented by directories in the file system. A map, hence, is represented by a directory containing all coverages. So before creating the UNGENERATE commands, the SQL query has to list the directory corresponding to the map. The algorithm for converting a global SQL subquery to access vector data of ARC/INFO maps is given below:

1. Generate the file system commands to link to the directory of the map. This directory is hence the **database** name concatenated with the **tablename**.

²A layer is called a coverage in ARC/INFO, but the converse may not be true.

2. Generate the command to invoke the ARC/INFO system. On most operating systems, this command is simply **arc**.
3. Generate the ARC/INFO command to list all the coverages (directories). On UNIX³ systems, this command is **LC**.
4. For each coverage, create an **UNGENERATE** command and assign the name of a temporary file to hold the results.
5. Generate the commands to invoke the translators between the **UNGENERATE**'s ASCII format and the Delta-X format.

It is much more difficult to translate an SQL query into GIS commands in some cases. For example, Caris does not provide a feature similar to **UNGENERATE**. So, the SQL query is translated into a command that downloads a Caris map into one of the interchange formats, say **CCOGIF**. If the Caris map contains topological information, the process becomes even more complicated. In this case, the map can only be downloaded into **CCOGIF** (or any other format) on a theme⁴ per theme basis. Finally, the SQL query generates the commands to invoke the Delta-X translators between **CCOGIF** and the Delta-X format. The algorithm for Caris is hence the following:

1. Generate the file system command to link to the directory of the map. In this case, the directory name is given in **database**. The map name refers to the actual Caris file.
2. Generate the command that returns the theme information including whether network and polygon topology have been created. On UNIX systems, this command is **dispthem**.
3. If the map has topology, then generate a **refoccssm** command (which converts into **CCOGIF**) for each theme number returned by **dispthem** and goto 5. Otherwise, goto 4.

³UNIX is a registered trademark of AT&T

⁴A theme in Caris corresponds to a layer

4. The map has no topology. One **refocsm** command is generated on the whole map.
5. Generate the commands to invoke the translators between the CCOGIF file(s) and the Delta-X format.

As stated in section 3.3, the Delta-X format is the interchange data format for GIS maps. Hence, the last step in translating an SQL query into GIS commands is always to generate or invoke a program that converts the given format (ASCII, CCOGIF etc) into Delta-X format. The actual conversion is performed at the DSI Gateway discussed in section 4.3.1. In short, the Delta-X [48] project is largely involved in defining the translators to and from the Delta-X format. These translators are used by the Local DBMS Translators of MQP.

Building such translators is simplified by the fact that the global SQL subqueries have a very simple form as shown previously. Clearly, building translators for the general SQL syntax can be prohibitively complex. This is our main justification for adopting the simple GDM and LDIs model.

4.2.3 DD/D and Auxiliary Schema Manager

The DD/D and Auxiliary Schema Manager is a component of the VDB Gateway, and hence of the Local DBMS Gateway. This means that the implementation of the DD/D and Auxiliary Schema Manager changes between Local DBMS Gateways. However, the services provided by the Manager are homogeneous regardless of the gateway in which it is implemented.

The DD/D and Auxiliary Schemas are stored under a separate database (user workspace, directory etc) under any one of the local DBMSs. If an RDBMS is available at the local site, then it is given priority for storing the DD/D and Auxiliary Schemas for obvious reasons. When the RDBMS supports databases, the DD/D and Auxiliary Schemas are stored under a database called MDBI DBDB. In any case, the DD/D and Auxiliary Schema Manager keeps track of the "database" under which these Schemas are stored.

2 OF 2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

Figure 4.1 DD/D Schema

Many MDBS, including MRDSM [68], do not separate the DD/D from the Auxiliary Schema. The same approach is followed by MDBI. In relational terminology, the DD/D and Auxiliary Schemas are a set of tables. The names and structures of these tables are fixed, regardless of the underlying local DBMS.

Figure 4.1 shows the ER diagram for the (mainly) DD/D component of MDBI_DBDB. The entity set **mdbi_db** represents databases, along with their DBMSs, site names and IP addresses that have been defined in one or more federations. Table and column definitions of tables defined in the federation are represented by **mdbi_tables** and **mdbi_columns** respectively. MQP uses the DD/D schema to get the qualifiers of tables and columns as discussed in section 4.2.2.

The entries in the DD/D are defined from the REGISTER commands (following EXPORT commands). Note that the same DD/D schema is replicated at all MDBI participating sites. To insure DD/D consistency, a transaction manager for the

EXPORT commands is needed.

This brings us back to the problems of multidatabase transaction management discussed in section 2.6. However, in this case, the following assumption can be made: The DD/D schema cannot be read nor written by local users. In fact, DD/D schema can only be read by the federation DBA and users, and can only be written by the federation DBA. Referring back to section 2.6.2, we see that in this special case the consistency of the DD/D schema can be guaranteed using 2PL/2PC and standard serializability since no local users are allowed to access this schema. That is, there are no local transactions on this schema. All transactions are global and hence under the control of the multidatabase transaction manager.

As part of mostly the Auxiliary Schema, MDBIDBDB defines two tables **localinfo** and **tasklog**. The structure of the table **localinfo** is shown below:

<i>Column name</i>	<i>Data Type</i>	<i>Description</i>
ipaddress	char(32)	IP address of the local site
hostname	char(32)	Name of local site
currtaskid	integer	Current value of task id
defaultdbms	char(32)	Default DBMS name
defaultdbname	char(32)	Default database name
defaulttimeout	integer	Default timeout for communications
defaulttasktimeout	integer	Default task timeout

There is only one entry in this table. This entry corresponds to the MDBI running at the local site. It is used to retrieve information relevant to task parameters.

Each global query is assigned with a site unique *taskid*. This task id is called the *initial* task id. An additional *fragment* task id is assigned to each global subquery of the initial global query. When prefixed by site name, the task ids become globally unique. In this case, they are referred to as global task ids (*gtaskid*). To insure that task ids are locally unique, **mdbi_localinfo** stores the current value of the task id. This value is incremented for each new task.

Task ids, in addition to default timeouts (if no timeouts are defined in the user interface) are assigned to the task parameter structure by the DD/D and Auxiliary Schema Manager.

The client component of the VDB uses the DD/D and Auxiliary Schema Manager to log each task that it creates. A task may correspond to the global query or to its subqueries. In either case, an entry is created for each task and will only be removed when the initial task has terminated. The structure of the `mdbi_logtask` Auxiliary Schema table is given below:

<i>Column name</i>	<i>Data Type</i>	<i>Description</i>
<code>gtaskid</code>	<code>char(80)</code>	Global task id of global query
<code>fgtaskid</code>	<code>char(80)</code>	Global task id of subquery
<code>remain</code>	<code>integer</code>	Number of remaining subqueries
<code>dbms</code>	<code>char(32)</code>	DBMS name
<code>dbname</code>	<code>char(32)</code>	Name of database
<code>tasktimeout</code>	<code>integer</code>	Task timeout used
<code>keepinmdb</code>	<code>integer</code>	Indicates if result is to be kept or deleted
<code>sender</code>	<code>integer</code>	Indicates if sender or receiver

Each task received by a server component is also logged until the task terminates and the results are sent back to the client.

It is possible to define additional information in the DD/D and Auxiliary Schema. A wide range of DD/D and Auxiliary schema information was presented in section 2.3.3.

4.3 The Remote DBMS Access Interface

The remote DBMS access interface has three layers. At the lowest layer is the Network Interconnection. RPC client and server routines are implemented at this level. They are used to handle the sending and receiving of one block of data between the host and the server machines. Above this layer is the Remote Network Access

Manager. It is responsible for establishing a connection with the server and for dividing the data into network blocks of adequate size that are iteratively sent to the NI layer for transmission. At the highest level is the Data Structure Interface. This interface is responsible for converting data between the format used by the local database and the standard format used for the interchange.

4.3.1 The Data Structure Interface

Different database systems and data models store their data in different formats. Furthermore, even within the same database, different data files, tables or objects have different definitions in terms of field/attribute names, types and domains. DSI defines a standard format, called the DSI Standard format, to provide a standard and consistent way of handling data transmitted over the network.

The DSI layer has the task of converting between the local database format and the DSI Standard. Although the DSI Standard is uniformly and consistently defined throughout the network, the local data representation is different at each site, depending on the representation of the local DBMS. Hence, DSI defines two main modules, the DSI Standard Representation and the DSI Gateway. The DSI Standard Representation module provides support for data structures used by participating databases. Notice first that all primitive data types (integers, float, characters) can be cast as strings of ASCII bytes. All complex data types (arrays, tables, objects) which themselves are built on top of primitive data types can also be cast as a stream of bytes, i. e., a text file. For example, a relation can be downloaded onto a text file. Even a complex object can be cast into a file by traversing it in a depth first search fashion. Of course, special characters have to be used as separators to indicate field, record, pointer or subobject boundaries. This being to allow the reconstruction of the original data at the destination site. Therefore, DSI has, at least in theory, the potential of providing support for any data structure. Practically, we will focus, as a first step, on the AD's defined in the ISO standards for SQL.

The DSI Standard follows the Standard Format Data Unit's [23] (SFDU) TLV model. In this model, an interchange format is expressed as TLV blocks. Each block

represents one data object. T represents the type of the object. L represents the length of the data value object. And, V represents the value of the object. MDBI defines in it Auxiliary Schema the standard types that it supports. These include integers, real numbers, strings, relational tables etc. The advantage of this scheme is that the DSI Standard becomes self-describing.

Because MDBI (RNAME and especially NI) is built on top of RPC/XDR, it is inherently portable across hardware systems, operating systems and network transports. DSI and the Local DBMS Gateway layers have the task of extending this portability across database systems and even data models. Where DSI and the Local DBMS Gateway intersect, we call DSI Gateway.

The DSI Gateway complements the DSI Standard Representation module in providing filters between the DSI standard format and the LDBMS format. At this stage of the project, we have mainly concentrated on relational database and simple flat file formats. The DSI Gateway routines, in the former case, have the task of working with the local DBMS in uploading and downloading relational tables into ASCII files.

Theoretically speaking, the MDBI should be fully portable to other data models. All that needs to be changed is the Local DBMS Gateway routines (including those of the DSI Gateway and the VDB Gateway). The difficulty here remains in converting data between two different data models (through the DSI standard) that have inherent mismatches. Take the case where the object-oriented model is included in the MDBI. Here, providing DSI routines for data transfer between two object-oriented DBMSs is not very difficult. The real difficulty is providing conversion routines between a relational table and a complex object. The mismatch is that while relational tables are flat in nature, complex objects are hierarchical. Similar problems are the mismatch of query languages at the VDB Gateway layer for language translators, and the semantic heterogeneity problems. In fact, these problems are equivalent to the problem of schema integration.

4.3.2 The Remote Network Access Manager

RNAM manages the network communications of NI. RNAM has two main functions depending on whether it is on the client or server side.

On the client's side, RNAM receives the data to be transmitted as a text file from the DSI layer. It also receives, from VDB the task parameters including a site unique task id. Recall that a task may correspond to either the initial global query or to one of its global subqueries. In the latter case, hence, a task refers to the sending of the text file over the network. If the size of the text file (data) is larger than the size of the maximum allowable block size, then RNAM will proceed to split the data into two or more blocks.

Each block is a record (i. e., a C - structure) containing several fields. The data is stored in a string field. In addition, other fields are used to store the global task id, block sequence number, a flag that indicates whether this is the last block for that task and an indication of the type of the block (SQL query, result relation... etc). The global task id is a network unique id for the current task. This field is obtained by concatenating the name of the host machine with the task id mentioned above.

Block type is defined as follows, using the RPC language syntax:

```
typedef string blkdatatype<>; /* definition for the data string */
struct blktype {
    blkdatatype blkdata;    /* data string in DSI Standard */
    unsigned    blkdatalen; /* length of data string */
    unsigned    blklen;     /* total block size in bytes */
    int         blkpos;     /* last block flag */
    int         blkno;      /* block sequence number */
    int         taskinfo;   /* type of task information */
    int         blkinfo;    /* subtype of block info */
    char        *host;      /* IP address of sender */
    char        *gtaskid;   /* unique global id for task */
}
```

```
char      *refgtid; /* reference gtaskid */
char      *dbms;    /* local DBMS of subquery */
char      *dbname;  /* local database of subquery */
/* may contain other fields */
}
```

The field `refgtid` refers to the global task id of the request query task in the case where the block contains result data for a query; `refgtid` is NULL otherwise. The string type `blkdatatype` is defined as a string of variable length and contains the data to be transferred in DSI Standard. Although theoretically TCP/IP allows for a maximum size of 4Gbytes, it is unlikely that this amount of space can be allocated for a string on any current machine. Hence, RNAM has also the task of determining the maximum possible amount of space that can be allocated to `blkdata` without overflowing up the heap. It should be noted that the bigger `blkdata` is, the less blocks it takes to transmit a task, the faster the total transmission of a task, and the higher the system throughput becomes.

Global tasks are defined in terms of a task type, represented by `taskinfo`. This field represents whether the task is an SQL query, result relation etc. Each task may have several subtasks. For instance, a result relation task has two subtasks: the result data itself, and the information used to recreate the table that stores this data. The latter are a list of the attributes and types of the result relation in conformance of the DSI Standard to the TLV model. Subtasks are identified by `blkinfo`. Therefore, while all the blocks of a global task must have the same `taskinfo`, `blkinfo` can be different between blocks of the same task depending on which subtasks they represent. Note how `blktype`'s are self-description, by also following the TLV approach.

There can be more than one file for each global task. These files are identified by the task id. The names of these (hidden) files have the following format `.AM.c.<gtaskid>.<blkinfo>`. The "c" represent the client side.

At the server's site, RNAM has the task of regrouping the blocks of a task into corresponding ASCII files. When the NI server at this site correctly receives a block,

it invokes RNAM before acknowledging the caller client. At this end, RNAM will create a file for each global task / blkinfo. RNAM uses the gtaskid field for creating file names which now have the format `.AM.s.<gtaskid>.<blkinfo>`. The RNAM protocol guarantees that the blocks are received in the same order they are sent. So RNAM only needs to append the blocks at the end of those files. Once the files are constructed (blkpos indicates the last block), RNAM invokes DSI with `gtaskid` as parameter.

4.3.3 The Network Interconnection Layer

NI is based on RPC TCP/IP for Internets. NI contains the RPC client and server programs as well as their supporting XDR filters, and client and server skeletons. The client and server routines are only responsible for the sending and receiving of one block of data.

NI clients always use the IP address when contacting the server. The client sends a `blktype` block. When the data is received by the server, it acknowledges the client by sending a reply message. This message only contains an indication of whether an error has occurred during the transmission as well as the error number.

At this level all that NI is concerned about is the correct sending of that block. The client and server routines are not aware nor concerned with the structure of the data block type. Neither is this layer aware of the existence and the structure of the layers above it. This modularity is highly desirable since it promotes flexibility and ease of integration of additional transport protocols. Hence, NI is not affected by any changes at the layers above it.

4.4 Local DBMS Gateways

RPC/XDR are used by the Remote DBMS Access Interface to provide support for hardware, communication and OS heterogeneity. The Local DBMS Gateways extend the heterogeneity support for DBMSs and data models. A Local DBMS Gateway has to be built for each DBMS (or GIS) product the MDBI wants to support.

The Local DBMS Gateways are interfaces to participating DBMSs, GISs and eventually file system applications. The MDBI is not permitted to access DBMSs (or GISs etc) directly. All access to DBMSs are done through the Local DBMS Gateways.

A site running the MDBI can have more than one Local DBMS Gateway. Each DBMS at the local site of an MDBI is associated with one of the Gateways. The Gateways, hence, should be easy to plug/unplug into/from the MDBI. For instance, if a new DBMS is installed at a given site, then its corresponding Local DBMS Gateway should be easy to install into the MDBI system running at that site. The new Gateway should not disturb the operations of the MDBI, nor any of the other Gateways.

The Local DBMS Gateways are the only components of the MDBI that are affected by the DBMSs and GISs running at the local site. All the other components of the MDBI are reusable across multiple DBMS and GIS platforms. As mentioned previously, the VDB Gateway and the DSI Gateway are part of the Local DBMS Gateways, but provide uniform functionalities regardless of the underlying DBMSs. This includes the DD/D and Auxiliary Schema Manager. Recall from section 4.2.2 that the DD/D and Auxiliary Schemas are stored under one of the local DBMSs and that their Manager is part of the VDB Gateway.

Gateways to non-SQL DBMSs, and specially to GIS products and file system applications have an extra layer that translates between the SQL model and the local model of the component DBS. For instance, Gateways to GIS databases use the Delta-X translators and the Delta-X format as an intermediate step between the local GIS and the MDBI as was discussed in sections 3.3 and 4.2.2. Eventually, such intermediate steps should be incorporated into the Auxiliary Schema to promote flexibility and reusability.

Local DBMS Gateways are general and do not expect any special kind of gateway support from the local DBMSs and GISs. The Gateways support full autonomy of the component DBSs and do not require rewriting of the components' software. Rather, the Gateways use whatever facilities the DBMSs and GISs provide for user

applications. Hence, a new Gateway has to be built for each individual DBMS and GIS product. However, much of the Gateway is reusable for DBMS that have the same data model.

Chapter 5

Current Prototype and Performance Model

In chapter 4, we presented the architecture of the MDBI. While the protocols that govern the way in which the different layers of the system interact may change over the life span of the Delta-X project, the layout of the system is expected to remain fairly stable.

The next section describes the current prototype of the MDBI. Section 5.2 discusses the current execution protocol of the system in the aim of establishing an analytical performance model.

5.1 Current Prototype

The implementation of a prototype MDBI is being carried out on a three site configuration connected by an Ethernet LAN at the Geographic Information Systems Division at Energy, Mines and Resources Canada (GISD at EMR). The configuration includes Caris and Ingres installations on a SUN¹ workstation, an Ingres installation on a SUN server, and an Oracle installation on an HP/UX². The system should be easily extendible to any UNIX/RPC platform. The test data includes two large databases (> 2 Gbytes) stored under Ingres on the SUN server.

¹SUN is a trademark of Sun Microsystems Inc.

²HP/UX is a trademark of Hewlett-Packard Co.

The MDBI prototype system is implemented in C. Its MQP component uses the LEX lexical analyzer generator and the YACC parser generator of UNIX. The Remote DBMS Access Interface uses the RPC/XDR facility for its Network Interconnection. The Local DBMS Gateway to Ingres uses Embedded C with Dynamic SQL, and the Gateway to Oracle uses Pro*C also with Dynamic SQL.

Oracle and Ingres provide their own gateways³ and seamless access to SQL/DS, DB2 and RDB. When these gateways are available, the MDBI would also be able to access SQL/DS, DB2 and RDB data indirectly.

The current configuration allows ad-hoc SELECT global queries to Ingres and/or Oracle. Other queries, including COPYDB, EXPORT and REGISTER have not yet been implemented.

Translators between the Delta-X format, and Caris, ARC/INFO (running on a fourth site), CCOGIF, DLG and SIF are available in the Delta-X project. Due to time constraints, these translators have not yet been incorporated with the Local DBMS Gateways of the MDBI.

The next step is to define the COPYDB and the federation administration commands and incorporate the system into the Delta-X project. This includes providing multidatabase interoperability to Caris and ARC/INFO, which will extend the configuration into four sites. There is also a fifth site that runs an Oracle installation on a Sun workstation. Furthermore, the Delta-X project will also provide the MDBI with an X11 user interface.

5.2 Performance Model

It should be clear at this point that the main objective of this thesis is to establish a proof of concept by implementing a working prototype of the MDBI proposal of DBMS/GIS interoperability. This thesis, hence, takes a practical approach instead of suggesting a system on paper and building a corresponding simulation model. Therefore, it would be more accurate to do performance modelling on the system

³In Oracle, these gateways are under the SQL*Connect subsystem.

itself, rather than on a simulation model that only approximates the behavior of the system.

The aim of this section is to analyse the performance of the MDBI by developing an analytical model. Again, due to time constraints, and because the main objective of this thesis is implementation rather than performance modelling, this section is only provided for the sake of completeness. The simplified queuing model of section 5.2.3 is developed mainly to provide some insights on the main potential problem areas in the performance of the MDBI. Such a model can be used as a basis for tuning the system as well as for developing a more general analytical model when performance issues are to be dealt with more accurately.

In general, global queries may be submitted to the MDBI which may be running at a large number of sites. Establishing the analytical model in the general case is hence quite complex. We restrict this section to a simplified MDBI environment.

Namely, we assume that, as in the prototype system, the MDBI is running at three sites only. Also, we do not consider the case of global queries, rather we assume that all queries are only remote queries. The latter restriction does not seriously limit the model since global queries are transformed into remote ones by MQP. Furthermore, it is expected that most GIS related queries are remote queries.

Recall from section 3.1.3 that the MDBI has a logical complete graph topology. Therefore, any of the three sites can send a remote query to any other site. With the lack of information about the actual environment in which the MDBI will be used, we will assume that it is equally likely that a site sends a query to any of the other two sites.

In the analytical model that we will develop, we use only Markovian queues with poisson arrivals, exponential service time and exponential interdeparture time. We assume that no failures can occur and that a remote query submitted for execution will eventually return with the results. Finally, we are only interested in the behavior of the system in the steady-state.

The next section describes in detail the execution protocol of remote queries. Section 5.2.2 presents some performance related issues. Equipped with this knowl-

edge, section 5.2.3 establishes the queuing model for the MDBI environment.

5.2.1 System Execution Protocol

Since global subqueries are expressed in SQL, then this section assumes that all remote queries are SQL queries. The execution of a remote query involves four steps: sending of the query to the participating site (and database), receiving the query by the participant, executing the query and returning of the results to the requesting site, and receiving the results by the initiator site.

5.2.1.1 Sending a Fragment Query

MQP of the VDB submits the query string to the DSI layer of the Remote DBMS Access Interface. It also submits the task parameters that include destination information.

The DSI normally has the task of converting a data structure into DSI standard, an ASCII file, and passes it along with the task parameters to RNAM.

RNAM creates a block of `blktype` for sending the query over the network. It assigns the query string to `blkdata`, sets `blkdatalen` and `blklen` appropriately, makes `blkpos` indicate that there are no more blocks in this task, sets `blkno` to 1, sets `taskinfo` and `blkinfo` to indicate an SQL query, assigns the `gtaskid`, and sets `refgtid` to NULL. Next, RNAM invokes the NI client. It also submits the created block and the task parameters to the client

The client routine of the NI creates an RPC client handle and sends the block to the NI server of the destination site. If the block is received correctly, the server will send an acknowledgment to the client. This acknowledgment travels back up through the layers of the remote DBMS access interface to the VDB. The latter will log the event and initiate a task level time-out process. This is done to make sure that a user does not have to wait indefinitely for the result of his query if the network link fails or if the participating site crashes after correctly receiving the query block.

5.2.1.2 Receiving the Query by the Participants

The NI server at the destination site forwards the received block to its RNAM. The latter extracts the query string and other fields from the blocks and passes them to the DSI layer. The query is stored in a file at this step. This file is submitted to VDB along with the task parameters. Included with these parameters are the values of **taskinfo** and **blkinfo**, which inform the VDB that the file corresponds to an SQL query.

The NI server acknowledges the client of the host while the VDB forks a new process to handle the execution of the query. This is done to allow increased concurrency between incoming requests. It also avoids the NI server from becoming the system's bottleneck.

5.2.1.3 Executing the Query and Returning the result

At the VDB, the query is submitted to the LDBMS for execution through the LDBMS Gateway. Currently, the LDBMS Gateway submits the query text to the LDBMS's embedded C - Dynamic SQL facility or the Local DBMS Translators for GIS's. The query is enriched by the gateway so that it creates a temporary table that holds the result of the execution. Next, the DSI Gateway downloads this table into a file.

Once a download file is created, the DSI submits this file, along with a file containing the recreation information of the table and task parameters received from the VDB, to RNAM. Among these parameters is **refgtid**. While the VDB will assign a new global task id to **gtaskid**, **refgtid** is assigned the **gtaskid** of the query block. This is done to enable the VDB of the initiator site to match the results returned with their request query. If the sizes of the files are larger than the maximum allowable size of a block, RNAM will split them into two or more blocks that are assigned the corresponding **blkinfo** values. Blocks are sent iteratively, and a block is only sent after the previous block transfer has been acknowledged.

The roles of the requesting site and the participating site are now reversed. The participating site becomes the client and the requesting site becomes the server of

the network communication. Hence, RNAME will invoke its NI client to send back the result to the initiator site. When all the RNAME blocks are acknowledged, the acknowledgment will travel back up all the way until the VDB.

5.2.1.4 Receiving the Result

NI server at the initiator site forwards the received blocks to RNAME. In addition to appending these blocks into files corresponding to the task/subtasks, RNAME also retrieves task parameters from the blocks. These include **taskinfo** and **blkinfo** which indicate that these blocks correspond to the result of a query, and **refgtid** which indicates the query to which the results belong. These are submitted to the DSI. The DSI Standard Representation module extracts the table recreation information and submits it to the DSI Gateway. The latter uses this information to recreate the table in the local database. The VDB now matches the **refgtid** with its log tables to determine which query has been satisfied and informs the user of the results.

Note that in this case also, the NI server of the initiator site acknowledges its client while a new process is again created to handle the DBMS related functions.

5.2.2 Performance Related Issues

There are two main performance determinants for the MDBI: the network communication, and the performance of the local DBMSs and their interfacing facilities.

The performance of the network communication, and the Remote DBMS Access Interface depends on the way the MDBI is built. The RPC/XDR routines at the NI level and RNAME have been built so as to optimize network communication. This is done by attempting to send blocks as large as possible so as to minimize the amount of overhead in repeated network transmissions. Since the number of network transmissions are minimized, the system throughput is maximized. The second performance issue is the parsing of a DSI Standard format (i.e an ASCII file) into a series of blocks and vice versa at the RNAME layer. Again, since the file is read and written in a per buffer basis (where the buffer size is set to the maximum),

the number of I/O operations are also minimized. Finally, note that having a DSI Standard representation has inherent drawbacks on performance. It represents an overhead especially when transferring data between two database systems of the same type and format. Yet, this overhead is worthwhile because it maximizes the portability of the MBDI. In the case of DBMSs, the time spent converting is expected to be insignificant compared to network overheads. The time required to move data down through the sender's protocol layers, across the network and up through the receiver's protocol, dwarfs the conversion overhead.

The other performance constraint falls entirely out of the MBDI control because it maintains the full autonomy of the local DBMSs. The MBDI gateways cannot access directly the data of the local database and has to use the database interfacing facilities.

The overhead of accessing local database data becomes quite substantial in the case of GISs. First of all, there is an additional overhead and indirection when accessing data from a GIS. As we saw in section 4.2.2, accessing data from (or to) ARC/INFO requires going through the UNGENERATE (or GENERATE) ASCII file format and the relational Delta-X format. Accessing data from Caris requires going through CCOGIF format and then converting between CCOGIF and Delta-X. Loading data into Caris requires "digitizing" the data by using CARED in batch mode. Hence, this also requires that a Delta-X map be translated into an ASCII file containing a series of CARED commands in addition to the data.

The MBDI can only attempt to use the most efficient interfacing facilities. For example, there are more than one command that can be used to load ASCII data into an ARC/INFO coverage (or vice versa). However, ARC/INFO manuals suggest that the GENERATE (or UNGENERATE) command is the most efficient.

Despite these measures, loading data into and out of a DBMS and specially a GIS is quite time consuming. In fact, this overhead is much more substantial than the communication cost in the case of local area networks.

Figure 5.1 Remote queries are executed in four steps

5.2.3 Analytical Model

First of all, recall from section 5.2 that the analytical model developed in this section is a means for providing insights on performance issues, rather than an end in itself. The main performance determinants of the MDBI running at each site is the network communication and the interfacing facility of the local DBMS or GIS. Therefore, each site is modelled by two queues corresponding respectively to the network communication and the LDBMS's interfacing facility.

Section 3.1.2 suggests that a remote query between two sites C_1 and C_2 is executed in two phases, the client phase and the server phase. Using the more detailed discussion of section 5.2.1, we remark that each phase consists of two steps. The first step relates to the network communication, and the second step relates to the local DBMS. Thus, the total number of steps for executing a remote query is four. The queuing network for a remote query that executes between two sites is shown in figure 5.1.

In this figure, ellipses represent service queues. If C_1 is the initiator site, the execution starts at queue 1 (exclusive) and ends at queue 1 (inclusive) following the sequence of arrows. Hence, the four steps are represented by the four queues.

Where queues 1 and 3 represent DBMS queues, and queues 2 and 4 represent network communication queues.

The two queues at each site cannot be merged into a single (possibly r -stage Erlangian) queue. This is due to the execution protocol of the MDBI. Each site has a single communication server. This server is modeled by queue 4 at site C_1 and queue 2 at site C_2 . Therefore, queues 2 and 4 are M/M/1 queues.

The MDBI server forks a new process to handle the data access from the DBMS for each request it receives. Therefore, queues 1 and 3 are multiple server queues. Under a very light system load, queues 1 and 3 can be thought of being M/M/ ∞ . In general, DBMSs and GISs cause a heavy load on their workstations. Although as many processes as needed are created to handle DBMS requests, the system load and hence the service time of these queues degrades as the number of processes increases. Thus, queues 1 and 3 are load-dependent queues with multiple servers.

Figure 5.2 generalizes this model to our case of three sites. In this figure, even-numbered queues are M/M/1 and odd-numbered queues are load-dependent queues with multiple servers.

Note that we are only representing network communication queues for MDBI servers, not for the clients. In the system's execution protocol presented in section 5.2.1, the MDBI client of the initiator site and the MDBI client of the participator site are both used when executing remote queries. In our simplified queuing model, we merge the service time of the initiator client with the service time of the participator's server. For example, if site C_1 is transmitting a query to site C_2 , then the whole cost of the network communication is assumed to be incurred at queue 2. This does not substantially affect our model because the assignment of queues to sites does not really affect the queuing model.

The same process that handles the DBMS access (odd-numbered queues) also becomes the client of the third step of the execution protocol. Therefore, the service time of this client is merged with the service time of the DBMS queue. For instance, if site C_2 is returning the results to site C_1 , then the service time of the client is included in queue 3.

Figure 5.2 Queuing network for the MDBI running at three sites

The two models shown previously serve only the purpose of identifying the main components involved in a remote query execution. The actual queuing model is developed shortly. In the general, we have an open network of queues. All odd-numbered (DBMS) queues will have arrows leaving to the outside world. Here, the "outside world" represents sites other than C_1 and C_2 . Also, all even-numbered (network communication) queues will have incoming arrows from the outside world. Therefore, our network of queues can be simplified to an open network of four queues. Here we assume that C_3 and potentially other sites is put back in the outside world. This situation is shown in figure 5.3.

The probability $p_{i,j}$ is defined as the probability for a customer (task) to move from queue i to queue j . Note that in our model, customers can only move between even and odd numbered queues; never between odd-numbered queues nor between even-numbered queues. The exception, of course, is the outside world which is represented by queue 0.

When a customer is serviced by an even numbered queue, the customer can

Figure 5.3 Open network of queues for the analytical model of MDBI

only move to an odd numbered queue with probability $p_{i,j} = 1$, where $i = 2n$ and $j = i+1$. Therefore, the probabilities for the open network of figure 5.3 are described by the following routing chain matrix.

$$\mathbf{P} = \begin{bmatrix} 0 & 0 & p_{0,2} & 0 & p_{0,4} \\ p_{1,0} & 0 & p_{1,2} & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ p_{3,0} & 0 & 0 & 0 & p_{3,4} \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \quad (5.1)$$

In the special case where there are only three sites in the MDBI environment, the outside world represents only site C_3 . We are assuming that it is equally likely that a site sends a customer to any of the other two sites. The probabilities in Eq. (5.1) are in this case equal to $1/2$. Therefore, the routing chain of Eq. (5.1) is given by

$$\mathbf{P} = \begin{bmatrix} 0 & 0 & 1/2 & 0 & 1/2 \\ 1/2 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1/2 & 0 & 0 & 0 & 1/2 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \quad (5.2)$$

Using the routing chains of Eqs. (5.1) and (5.2), we can calculate the relative throughputs assuming poisson arrivals from the outside world with parameter λ_0 . Solving the equation $\lambda \mathbf{P} = \lambda$ for Eq. (5.1) gives

$$p_{1,0}\lambda_1 + p_{3,0}\lambda_3 = \lambda_0 \quad (5.3)$$

$$\lambda_4 = \lambda_1 \quad (5.4)$$

$$p_{0,2}\lambda_0 + p_{1,2}\lambda_1 = \lambda_2 \quad (5.5)$$

$$\lambda_2 = \lambda_3 \quad (5.6)$$

$$p_{0,4}\lambda_0 + p_{3,4}\lambda_3 = \lambda_4 \quad (5.7)$$

These equations can now be solved for the throughput of each queue in terms of λ_0

$$\lambda_1 = \frac{1 - p_{3,0}p_{2,0}}{p_{1,0} + p_{3,0}p_{1,2}} \lambda_0 \quad (5.8)$$

$$\lambda_2 = \left(p_{0,2} + p_{1,2} \frac{1 - p_{3,0}p_{2,0}}{p_{1,0} + p_{3,0}p_{1,2}} \right) \lambda_0 \quad (5.9)$$

$$\lambda_3 = \left(p_{0,2} + p_{1,2} \frac{1 - p_{3,0}p_{2,0}}{p_{1,0} + p_{3,0}p_{1,2}} \right) \lambda_0 \quad (5.10)$$

$$\lambda_4 = \frac{1 - p_{3,0}p_{2,0}}{p_{1,0} + p_{3,0}p_{1,2}} \lambda_0 \quad (5.11)$$

To solve for the case of three sites, we substitute the probability values by 1/2

$$\lambda_1 = \lambda_0 \quad (5.12)$$

$$\lambda_2 = \lambda_0 \quad (5.13)$$

$$\lambda_3 = \lambda_0 \quad (5.14)$$

$$\lambda_4 = \lambda_0 \quad (5.15)$$

$$(5.16)$$

Because we are using poisson interarrivals, exponential service time Markovian queues, we can use the relative throughputs to solve for each queue individually. We solve these queue as birth-death systems in equilibrium since we are only interested in the steady state behavior. For general birth-death systems [32-34, 45], π_k is given by

$$\begin{aligned} \pi_k &= \left(\prod_{j=0}^{k-1} \frac{\lambda_j}{\mu_{j+1}} \right) \pi_0 \\ \sum_{k=0}^{\infty} \pi_k &= 1 \\ \pi_0 &= \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{j=0}^{k-1} \frac{\lambda_j}{\mu_{j+1}}} \\ \pi_k &= \frac{\prod_{j=0}^{k-1} \frac{\lambda_j}{\mu_{j+1}}}{1 + \sum_{k=1}^{\infty} \prod_{j=0}^{k-1} \frac{\lambda_j}{\mu_{j+1}}} \end{aligned} \quad (5.17)$$

There are two types of queues in our system, M/M/1 (even-numbered) numbered and Load-Dependent Multiple Server (odd-numbered) queues. Therefore, it is sufficient to solve for one odd-numbered queue i , say queue 1; and solve for one even-numbered queue i , say queue 2. The results can then be generalized to the other queues.

Let queue i be a Load-Dependent Multiple Server queue. Therefore, the service time of queue i degenerates as the number of customers increase. The rate of decrease in service time depends on many factors. These factors include the type of hardware of the local site, the type of operating system and its virtual memory protocols, and mostly the type of DBMS(s) and/or GIS(s) that are running (simultaneously) at the local site. These three main factors are also interdependent. For instance, the service time and performance degradation of a DBMS depends on the type of hardware and OS it is running on. Also related to DBMS and GIS performance is the type of queries and commands that are being executed. For example, a query

involving joins takes a considerable time to execute and uses much more resources than a query that only performs selections and projections.

A high system load relates to the number of customers as well as the hardware, OS, DBMSs and GISs. For instance, a very high system load is not only due to a large number of customers, but also to the number of DBMS and GIS processes and the system load from other non-MDBI related processes. Therefore, it is virtually impossible to accurately compute the rate of performance degradation in general.

One way to overcome this is to use the statistical data of the vendor to approximate this figure. In the absence of such data, we will assume that the system degrades by an unknown quantity. This quantity is clearly a function of k , $f(k)$ where k is the number of customers. Hence, for (odd-numbered) queues i , the service rate is given by $\mu_i(k) = k\mu_i/f(k)$. Let π_k be the probability of finding k customers in a queue. Using Eq. (5.17) we get the following

$$\pi_k = \frac{\prod_{j=0}^{k-1} \frac{\lambda_i}{(j+1)\mu_i/f(j+1)}}{1 + \sum_{k=1}^{\infty} \prod_{j=0}^{k-1} \frac{\lambda_i}{(j+1)\mu_i/f(j+1)}} \quad (5.18)$$

It is expected that under light load (small number of customers), the rate of degradation is negligible. As the number of customers increase, performance is expected to degrade by a polynomial function of k in n , $f(k) = O(k^n)$ where n increases with system load. For very high load, the rate of degradation may even become exponential.

In the case of very light system load, $f(k)$ reduces to a constant and hence odd-numbered queues can be approximated by M/M/ ∞ queues. The case of very high system load is very complex since it involves an exponent in k . In the current three site configuration of the system, the case of very high system load case is highly unlikely. Even in the general case, workstations are not expected to have more than one or two DBMSs and/or GISs. In fact, the only sites that are expected to have high loads are server systems. However, these systems usually have powerful processors which balance the increase in system load.

Therefore, we consider the case where the load is medium, yet low enough that $f(k)$ is linear. That is, $f(k) = O(k) \cong Ck$ where C is a constant. To simplify

the model we have assumed that the parameters for the hardware, OS, DBMSs and GISs are constant and have been incorporated into C . Hence, the service rate for odd-numbered queues i becomes $\mu_i(k) = k\mu_i/Ck = \mu_i/C$. Substituting in Eq. (5.18) gives the following, where $\lambda_i C < \mu_i$.

$$\begin{aligned}\pi_k &= \left(\frac{\lambda_i C}{\mu_i}\right)^k \pi_0 \\ \pi_0 &= \frac{1}{1 + \sum_{k=1}^{\infty} \left(\frac{\lambda_i C}{\mu_i}\right)^k} \\ \pi_0 &= 1 - \frac{\lambda_i C}{\mu_i} \\ \pi_k &= \left(1 - \frac{\lambda_i C}{\mu_i}\right) \left(\frac{\lambda_i C}{\mu_i}\right)^k\end{aligned}\quad (5.19)$$

The system utilization ρ_i of queue i is given by

$$\begin{aligned}\rho_i \equiv 1 - \pi_0 &= \frac{\lambda_i C}{\mu_i} \\ \pi_k &= (1 - \rho_i) \rho_i^k\end{aligned}\quad (5.20)$$

These results are similar as those of M/M/1 queues to within a constant factor. In fact, these results justify our choice of a linear rate of performance degradation. The local site is really a single server queue. As in time sharing systems, concurrency only promotes the increase of turnaround time. That is, it increases the number of tasks that are completed in a unit of time. But it does not increase the performance of any task. In the case of a reasonable system load, the total service time of the tasks is expected to be the same as in the case of a single server system to within a constant factor; the difference is that the tasks have been interleaved. The constant factor is due to the overhead of concurrency and hence depends on the underlying time-sharing system. Thus, these results conform with our model. The constant is in our case C which depends on the factors mentioned previously.

The average number of customers N in queue i is given by [33, 34, 45]

$$N_i = \frac{\rho_i}{1 - \rho_i} \quad (5.21)$$

Using Little's result, we can calculate the average delay T_i

$$\begin{aligned} N_i &= \frac{\rho_i}{1 - \rho_i} \\ &= \lambda_i T_i \\ T_i &= \frac{C/\mu_i}{1 - \rho_i} \end{aligned} \quad (5.22)$$

To solve for queue 1 (or 3) we need only replace the values of λ_i and μ_i with λ_1 and μ_1 . For the case of three sites, queue 1 has an interarrival rate of $\lambda_1 = \lambda_0$.

Let queue i be an even-numbered queue, then queue i is M/M/1. Eq. (5.17) gives, where $\lambda_i < \mu_i$

$$\begin{aligned} \pi_k &= \left(\frac{\lambda_i}{\mu_i} \right)^k \pi_0 \\ \pi_0 &= \frac{1}{1 + \sum_{k=1}^{\infty} \left(\frac{\lambda_i}{\mu_i} \right)^k} \\ \pi_0 &= 1 - \frac{\lambda_i}{\mu_i} \\ \pi_k &= \left(1 - \frac{\lambda_i}{\mu_i} \right) \left(\frac{\lambda_i}{\mu_i} \right)^k \end{aligned} \quad (5.23)$$

The system utilization is hence given by

$$\begin{aligned} \rho_i \equiv 1 - \pi_0 &= \frac{\lambda_i}{\mu_i} \\ \pi_k &= (1 - \rho_i) \rho_i^k \end{aligned} \quad (5.24)$$

Using Little's result, we get the average number of customers and the average delay time for customers in queue i

$$\begin{aligned} N_i &= \frac{\rho_i}{1 - \rho_i} \\ &= \lambda_i T_i \\ T_i &= \frac{1/\mu_i}{1 - \rho_i} \end{aligned} \quad (5.25)$$

Using these results, we get the average number of customer waiting, and the average waiting time in queue i

$$N_{qi} = \frac{\rho_i^2}{1 - \rho_i}$$

$$W_i = \frac{\lambda_i / \mu_i^2}{1 - \rho_i} \quad (5.26)$$

To solve for queue 2 (or 4), we need only replace λ_i with λ_2 and μ_i with μ_2 . Again, in the case of the three site configuration, $\lambda_2 = \lambda_0$.

The infinite server property of the Load Dependent Multiple Server queues is not desirable when a participant is expected to have a large number of concurrent DBMS processes. In this case, the system would essentially be thrashing. One way to solve this problem is to make the DBMS queues become finite server queues (M/M/m), instead of infinite server queues. In this case, the system will block additional customers DBMS queues to avoid overloading these queues and hence the work-load of their local workstation. Therefore, this scheme avoids performance degradation. Of course, the value of m would have to be computed. This value should achieve a trade-off between lower loads and higher concurrency. It depends on the local system and therefore on $f(k)$, or C in the simplified case.

Validation of this analytical model could not be performed due to the lack of information and resources. As we have mentioned previously, vendor statistics on the service times for the DBMSs (Ingres and Oracle) and GISs (ARC/INFO and Caris) are not currently available to us. Also, the number and type of queries that are expected to run on the system have not been determined from the potential users of the system. This means that the values of μ_i and λ_0 cannot, currently, be computed. Also, test results from the current MDBI prototype cannot be conclusive since many of the factors affecting service time (system workloads including service daemons, remotely logged users etc) cannot be controlled without disturbing the working environment of the GISD at EMR. Moreover, DBMS and GIS resources are not available for experimentation at the School of Computer Science at Carleton University. The future work on the MDBI, hence, includes the validation of the analytical model when the required information and resources are available.

Chapter 6

Conclusion

Several data models for database systems have been proposed over the past few decades. For each of these data models, numerous DBMS products have been developed and are now commercially widespread. These systems are essentially isolated and provide limited or no interoperability between themselves and others. This has resulted in what has come to be known as the islands of information problem. Multidatabase have been proposed to solve this problem.

In an MDBS environment, the data might be distributed over the network. In addition, data might also be partitioned and replicated. This distribution usually predates the MDBS, so the MDBS has to cope with it.

The MDBS should not interfere with the local operations of the component DBMSs. This is known as the requirement for autonomy of the component DBMSs in a multidatabase environment. Hence, the requirement for autonomy assumes that no changes can be made to the participating DBMS software. The requirement for autonomy of component DBMSs causes serious problems for query processing and specially transaction management. Transaction management is still an open area of research. The difficulties in transaction management arise because the MDBS does not have control over local transactions. There are two main approaches. The first approach develops transaction management models that maintain the traditional 2PL/2PC serializability, but impose various restrictions on the multidatabase environment. The second approach proposes new serializability models which are

better suited for multidatabases environments. Many researchers argue that the latter approach is more promising.

Heterogeneity arises for many reasons. Among them result the following two types of heterogeneity: heterogeneity due to differences in DBMSs and semantic heterogeneity. The latter problem is the more difficult one. It is still not a well defined and well understood problem. It is an active area of research in multidatabases.

The main goal of MDBSs is to provide component DBMSs interoperability. This interoperability has been addressed by DBMS vendors and MDBS research prototypes. But, often their approach is either custom tailored to specific systems or data models, or imposes various restrictions on the MDBS environment. Typically, the DBMS vendors approach requires major rewriting to the DBMS software to allow interoperability.

6.1 Main Results and Contributions

The work presented in this thesis is the MultiDataBase Integrator that acts as a backbone loosely coupled multidatabase system. The following are the main results and contributions developed in this thesis:

- The thesis is mainly concerned with defining the architecture of the MDBI and implementing a working prototype.
- A multidatabase extension to SQL, MDBI*SQL, custom tailored to support GIS application development, is also developed.
- MDBI, with its MDBI*SQL query language, is proposed for distributed interoperability, not only between DBMSs, but also between DBMSs and/or GISs simultaneously. This proposal is unique.
- MDBI defines a variant of the loosely coupled federated architecture in the multidatabase taxonomy.
- MDBI defines a client/server model for remote query execution.

- The implemented prototype supports global distributed MDBI*SQL queries between Ingres and Oracle RDBMSs.
- A simplified queuing model is provided to give some insights to some of the main performance issues.
- Finally, we present a general framework for multidatabase access applicable to other applications such as scientific databases.

The MDBI has three main components: the Virtual DataBase, the multidatabase manager of MDBI; the Remote DBMS Access Interface, the network communication component of the MDBI; and the Local DBMS Gateways, that provide homogeneous interface to heterogeneous DBMSs and GISs.

The Virtual DataBase is the top level of the MDBI. It defines the MDBI*SQL language by implementing a Multidatabase Query Processor. This Processor is used to split and fragment a global MDBI*SQL query into SQL global subqueries that can be executed by a single database at a single site. Such subqueries are also termed remote queries. The Processor also defines the Local DBMS Translators, that are used to translate an SQL subquery into equivalent GIS commands for map data access.

The Virtual DataBase also defines the system's Data Dictionary/Directory and Auxiliary Schemas. These system catalogs are used to store information about the participating databases in the MDBI. This information include the site names, the DBMS types installed at these sites and the databases stored in these DBMSs. Information about the databases include database names, component tables (files, objects) and their definitions and domains. All this information is used by the query processor to split and fragment a global query into corresponding global subqueries. The information is also used to send the subqueries into the participating DBMSs.

Auxiliary databases are also used to support language as well as data model heterogeneity. The goal is to make the ISO standard SQL the *canonical* language for the MDBI. The standard is currently at SQL2, which provides limited support for object-orientation and Abstract Data Types. SQL3, an Object-Oriented SQL is

being drafted for standardization by ISO. Auxiliary databases can also be used to assist in heterogeneity issues that arise due to differences in data models. Semantic heterogeneity is one of such issues.

The Remote DBMS Access Interface is subdivided into three main layers: the Network Interconnection, the Remote Network Access Manager and the Data Structure Interface. Together, these three layers are responsible for handling the distributed component of the MDBI. They are also used to handle the data structures transmitted over the network by defining a DSI interchange format. The Remote DBMS Access Interface defines a client/server model for the execution of remote SQL queries.

The Local DBMS Gateways are gateways of the MDBI to the DBMSs and GIS running at the local site. The Gateways are the only component of the MDBI that are affected by the local DBMSs and GISs. However, these gateways are designed to provide a homogeneous interface to heterogeneous DBMS and GIS products. The Gateways respect the full autonomy of the local database systems, and interface to these systems as if gateways were user applications. Currently, Gateways to Ingres and Oracle have been implemented. Also, pilot implementations for translators to/from ARC/INFO and CARIS have also been carried out.

6.2 Future Directions

The current limitations of the system is that it does not yet implement the COPYDB, EXPORT and REGISTER commands. Only global SELECT queries are currently supported. Also, the system does not implement any fault-tolerance capabilities. This is not, currently, a very serious limitation since the MDBI is a retrieval only system at this point.

Future work on the MDBI consists mainly of validating the analytical model, implementing the COPYDB, EXPORT and REGISTER commands, integrating the MDBI with the Delta-X project and eventually implementing a multidatabase trans-

action management. MDBI*SQL can also be extended to implement more multidatabase features such as those defined in the MRDSM project.

The implementation of the REGISTER command requires a multidatabase transaction manager. However, it can be assumed that no local transactions are allowed on the DD/D schema. This is a special case of section 2.6.2. Hence, a distributed 2PL/2PC and serializability guarantees the consistency of the DD/D.

In addition, the MDBI can be used as a testbed for further experimentation on the main open areas for research the multidatabase field. These include semantic heterogeneity in database integration, query optimization, and transaction processing. Referring back to section 2.6.1, note how the transaction models proposed in the literature limit severely the concurrency and interleaving between global sub-transactions. The MDBI offers a good starting base for benchmarking these various algorithms, and establishing their practicality.

The integration of the MDBI into the Delta-X project includes the integration of GIS translators into Local DBMS Gateways to implement the proposed distributed GIS interoperability with MDBI*SQL. Global optimization protocols can also be implemented into the Multidatabase Query Processor. A transaction manager (concurrency control, commit protocols and crash recovery) could be implemented at the Virtual DataBase level if and when a multidatabase transaction management model is defined or adopted.

For the longer term, an X11 interface that also allows map display is a highly desirable feature for the MDBI project. Also, the federated data model can be redefined as object-oriented instead of relational. This opens up many research issues including an unprecedented mixture of object-oriented and multidatabase extensions to MDBI*SQL.

Bibliography

- [1] R. Alonso and D. Barbara. Negotiation data access in federated database systems. In *Proc. 5th Int. Conf. on Data Engineering*, pages 56 – 65, February 1989.
- [2] K. Barker and M. T. Ozsü. Concurrent transaction execution in multidatabase systems. In *Proc. COMPSAC'90*, Chicago, Illinois, October 1990.
- [3] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Comput. Surv.*, 18(4):323 – 364, December 1986.
- [4] Y. J. Breitbart and A. Silberschatz. Multidatabase update issues. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 135 – 142, Chicago, June 1988.
- [5] Y. J. Breitbart, A. Silberschatz, and G. R. Thompson. Transaction management in a multidatabase environment. In A. Gupta, editor, *Integration of Information Systems: Bridging Heterogeneous Databases*, pages 135 – 143. IEEE Press, New York, 1989.
- [6] Y. J. Breitbart, A. Silberschatz, and G. R. Thompson. Transaction management issues in a failure-prone multidatabase system environment. *Very Large Data Bases Journal*, 1(39), 1992.
- [7] Y. J. Breitbart, A. Silberschatz, and G. R. Thompson. An update mechanism for multiple database systems. *Data Eng.*, 10(3):12 – 18, September 1987.

- [8] D. Brill, M. Templeton, and C. Yu. Distributed query processing strategies in mermaid: A front-end to data management systems. In *Proc. First Int. Conf. on Data Engineering*, pages 211 – 218, Los Angeles, 1984.
- [9] Z. Brzezinski, J. Getta, J. Rybnik, and W. Stepniewski. Unibase: An integrated access to database. In *Proc. 10th Int. Conf. on Very Large Data Bases*, pages 388 – 400, Singapore, August 1984.
- [10] A. F. Cardenas. Heterogeneous distributed database management: Hd-dbrms. *Proc. IEEE*, 75(5):588 – 600, May 1987.
- [11] S. Ceri and G. Pelagatti. *Distributed Databases - Principles and Systems*. McGraw-Hill, New York, 1984.
- [12] N. Cerone, M. Morgestern, A. Sheth, and W. Litwin. Resolving semantic heterogeneity. In *Int. Conf. on Data Engineering*, February 1990.
- [13] A. L. P. Chen, D. Brill, M. P. Templeton, and C. T. Yu. Distributed query processing in mermaid: A frontend system for multiple database. *IEEE J. Selected Areas Commun.*, 7(3):390 – 398, April 1989.
- [14] D. Cheriton. Vmtp: Versatile message transaction protocol. Technical report, Stanford University, January 1987. preliminary version 0.3.
- [15] B. Demo. Program analysis for converison from a navigational to a specification data base. In *Proc. 9th Int. Conf. on Very Large Data Bases*, pages 387 – 398, Florence, Italy, October 1983.
- [16] W. Du and A. K. Elmagarmid. Quasi-serializability criterion for global concurrency control in interbase. In *Proc. 15th Int. Conf. on Very Large Data Bases*, pages 347 – 355, Amsterdam, August 1989.
- [17] W. Dunquaine. Mainframe dbms connectivity via general client / server approach. *IEEE Data Engineering Bulletin*, 13(2), 1990.

- [18] P. Dwyer and J. L. Larson. Some experiences with a distributed database testbed system. *Proc. IEEE*, 75(5):633 - 648, May 1987.
- [19] F. Eliassen and J. Veijalainen. A functional approach to information system interoperability. In R. Speth, editor, *Research into Networks and Distributed Applications (Proceedings of the EUTECO '88)*, pages 1121 - 1135. Elsevier Science Publishers B.V., North-Holland, 1988.
- [20] R. Elmasri, J. Weeldreyer, and A. Hevner. The category concept: An extension to entity-relationship model. In *Data and Knowledge Engineering*, pages 75 - 116. North-Holland, The Netherlands, June 1985.
- [21] Environmental Systems Research Institute, Inc. *ARC/INFO Data Models, Concepts, & Key Terms, ARC/INFO users Guide*, 1991. Version 6.0.
- [22] A. Ferrier and C. Stangret. Heterogeneity in the distributed data management system sirius-delta. In *Proc. 8th Int. Conf. on Very Large Data Bases*, pages 45 - 53, Mexico City, September 1982.
- [23] Consultative Committee for Space Data Systems. *Standard Formatted Data Units: Structure and Construction Rules. Red Book, Issue 2*. National Aeronautics and Space Administration, February 1987.
- [24] L. Gallagher and J. Sullivan. Database language sql: Integrator of cals data repositories, November 1991. ISO/IEC JTC1/SC21/WG3 DBL KAW-98-26 DRAFT.
- [25] G. Gardarin and P. Valduriez. *Relational Databases and Knowledge Bases*. Addison-Wesley, Reading, Mass., 1989.
- [26] V. D. Gligor and G. L. Luckenbaugh. Interconnecting heterogeneous database management systems. *IEEE Computer*, 17(1):33 - 43, January 1984.
- [27] V. D. Gligor and R. Popescu-Zeletin. Transaction management in distributed heterogeneous database management systems. *Inf. Syst.*, 11(4):287 - 297, 1986.

- [28] ISO. Iso/tc97: Draft international standard iso/dis 7489: Information processing systems-open systems interconnection-basic reference model, 1982.
- [29] ISO. Remote database access protocol. 2nd working draft, 1987. ISO/TC 97/SC 21/WG 3.
- [30] B. Jacobs. *Applied Database Logic II: Heterogeneous Distributed Query Processing*. Prentice-Hall, Englewood Cliffs, NJ, 1985.
- [31] G. Jacobson, G. Piatetsky-Shapiro, C. Lafond, M. Rajinikanth, and J. Fernandez. Calida: A knowledge-based system for integrating multiple heterogeneous databases. In *Proc. 3rd Int. Conf. on Data and Knowledge Bases*, pages 3 – 18, June 1988.
- [32] E. G. Coffman Jr. and P. J. Denning. *Operating Systems Theory*. Prentice-Hall, 1973.
- [33] L. Kleinrock. *Queueing Systems*, volume 1. John Wiley & Sons, Inc., New York, 1975.
- [34] L. Kleinrock. *Queueing Systems*, volume 2. John Wiley & Sons, Inc., New York, 1976.
- [35] E. Levy and A. Silberschatz. Distributed file systems: Concepts and examples. *ACM Comput. Surv.*, 22(4):321 – 374, December 1990.
- [36] W. Litwin. Malpha: A relational multidatabase manipulation language. In *1st IEEE Conference on Data Engineering*, Los Angeles, Calif., February 1984.
- [37] W. Litwin. Implicit joins in the multidatabase system mrds. *IEEE-COMPSAC*, pages 495 – 504, 1985.
- [38] W. Litwin. An overview of the multidatabase system mrds. In *Proc. ACM National Conference*, pages 495 – 504, Denver, October 1985.

- [39] W. Litwin and A. Abdellatif. Multidatabase interoperability. *Computer*, 19(12):10 – 18, December 1986.
- [40] W. Litwin and A. Abdellatif. An overview of the multidatabase manipulation language-mdsl. *Proc. IEEE*, 75(5), May 1987.
- [41] W. Litwin, A. Abdellatif, B. Nicolas, P. Vigier, and A. Zeroual. Msq: A multidatabase language. Technical report, INRIA, BP 105, 78153 Le-Chesnay, France, 1987. 695.
- [42] W. Litwin, J. Baudenant, C. Esculier, A. Ferrier, A. M. Glorieux, J. LaChimia, K. Kabbaj, C. Moulinoux, P. Rolin, and C. Stangret. Sirius system for distributed data management. In H.-J. Schneider, editor, *Distributed Data Bases*, pages 153 – 184. North-Holland, Amsterdam, 1982.
- [43] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Comput. Surv.*, 2(3), September 1990.
- [44] W. Litwin and A. Zeroual. Advances in multidatabase systems. In R. Speth, editor, *Research into Networks and Distributed Applications (Proceedings of the EUTECO '88)*, pages 1137 – 1151. Elsevier Science Publishers B.V., North-Holland, 1990.
- [45] M. K. Molloy. *Fundamentals of Performance Modeling*. Macmillan Publishing Company, New York, 1989.
- [46] E. J. Otoo. The design and implementation of a prototype gis database for application development. Technical report, Energy, Mines and Resources Canada, 1992. Summary Report.
- [47] E. J. Otoo and M. M. Allam. Spatial topological model: A first step towards gis data exchange standard. Technical report, Energy, Mines and Resources Canada, 1991.

- [48] E. J. Otoo, A. Mamhikoff, L. Ying, and C. Chaly. Browsing and interoperability of federated databases for geographic information systems. In *Fifth Int. Conf. on Geographic Information Systems (The Canadian Conference on GIS)*, Ottawa, Canada, March 1993.
- [49] M. T. Ozsü and P. Valduriez. *Principles of Distributed Databases*. Prentice Hall, Englewood Cliffs, NJ, 1991.
- [50] J. Postel. User datagram protocol. Technical report, Information Sciences Institute, August 1980. RFC 768.
- [51] J. Postel. Transmission control protocol - darpa internet program protocol specification. Technical report, Information Sciences Institute, September 1981. RFC 793.
- [52] J. S. Quarterman and J. C. Hoskins. Notable computer networks. *Communications of the ACM*, 29(10), October 1986.
- [53] S. Ram and C. Chastain. Architecture of distributed data base systems. *J. Syst. Softw.*, 10(2):77 – 95, 1989.
- [54] Relational Technologies, Inc. *Distributed Ingres*, November 1989. release 6.3.
- [55] Relational Technologies, Inc. *Ingres/SQL Reference Manual*, January 1990. release 6.3, UNIX.
- [56] Relational Technologies, Inc. *Ingres Open SQL Reference Manual*, December 1991. release 6.4.
- [57] M. Rusinkiewicz and B. Czejdo. Query transformation in heterogeneous distributed database systems. In *Proc. 5th Conf. on Distributed Computing Systems*, Denver, Colorado, May 1985.
- [58] M. Rusinkiewicz, R. Elmasri, B. Czejdo, D. Geogakopoulos, G. Karabatis, A. Jamoussi, K. Loa, and Y. Li. Omnibase: Design and implementation of a multi-

- database system. In *Proc. First Annual Symp. Parallel and Distributed Computing*, pages 162 – 169, Dallas, Texas, 1989.
- [59] M. Rusinkiewicz, R. Elmasri, B. Czejdo, D. Geogakopoulos, G. Karabatis, A. Jamoussi, K. Loa, Y. Li, J. Gilbet, and R. Musgrove. Query processing in omnibase-a loosely coupled multi-database system. Technical report, Department of Computer Science, University of Houston, Houston, Texas, February 1988. UH-CS-88-05.
- [60] A. Sheth and S. Gala. Attribute relationships: An impediment in automating schema integration. In *Workshop on Heterogeneous Database Systems*, Chicago, Ill., December 1989.
- [61] A. Sheth and J. L. Larson. Federated databases systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv.*, 2(3), September 1990.
- [62] A. Sheth, J. L. Larson, A. Cornellio, and S. B. Navathe. A tool for integrating schemas and user views. In *Proc. 4th Int. Conf. on Data Engineering*, pages 176 – 183, Los Angeles, February 1988.
- [63] D. Simonson and D. Benningfield. Ingres gateways: Transparent heterogeneous sql access. *IEEE Data Engineering Bulletin*, 13(2), 1990.
- [64] SUN Microsystems, Inc. *Network Programming Guide*, March 27 1990. revision A.
- [65] M. Templeton, D. Brill, S. K. Dao, E. Lund, P. Ward, A. L. P. Chen, and R. McGregor. Mermaid - a front-end to distributed heterogeneous database. *Proc. IEEE*, 75(5):695 – 708, May 1987.
- [66] G. Thomas, G. R. Thompson, C.-W. Chung, E. Barkmeyer, F. Carter, M. Templeton, S. Fox, and B. Hartman. Heterogeneous distributed database systems for production use. *ACM Comput. Surv.*, 2(3), September 1990.

- [67] J. Veijalainen and R. Popescu-Zeletin. Multidatabase systems in iso/osi environment. In *Standards in Information Technology and Industrial Control*, pages 83 - 97. North-Holland, The Netherlands, 1988.
- [68] K. K. Wong and P. Bazez. Mrdsm: A relational multidatabase management system. In F. A. Schreiber and W. Litwin, editors, *Distributed Data Sharing Systems*, pages 77 - 85. North-Holland, Amsterdam, 1985.
- [69] Xerox. Courier: The remote procedure calls. Technical report, Xerox Corporation, December 1981. XSIS 038112.

END

14-12-93

FIN