# INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**University of Alberta**

# Building Scalable and Flexible Mediation: the AURORA Approach

by

**Ling Ling Yan**

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Spring 2000

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

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

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

0-612-60042-4

Canadä

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**University of Alberta**

**Library Release Form**

**Name of Author:** Ling Ling Yan

**Title of Thesis:** Building Scalable and Flexible Mediation: the AURORA Approach

**Degree:** Doctor of Philosophy

**Year this Degree Granted:** 2000

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

. . . . . . . . . . . . . . . . . . . . . .
Ling Ling Yan
364 Spode Way
San Jose, California
United States, 95123

**Date:** Jan 27, 2000

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Everything should be as simple as possible – but not simpler.
– Albert Einstein

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**University of Alberta**

**Faculty of Graduate Studies and Research**

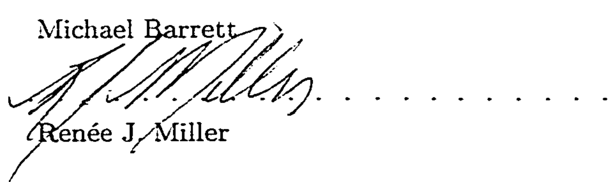
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Building Scalable and Efficient Mediation: the AURORA Approach** submitted by Ling Ling Yan in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**

M.Tamer Özsu

Li Yan Yuan

Duane Szafron

Michael Barrett



Renée J. Miller

**Date:** Jan 27, 2000

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To Murray

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# Abstract

This dissertation describes the AURORA project, that develops approaches and techniques for large-scale data integration. The focus of the project is on the following:

1. *Scalable mediation.* Adding and removing data sources to/from the access scope of a data integration system should be easy.

2. *Flexible mediation.* Specific properties of data sources, such as data models, query processing capabilities, and availability, should be dealt with by the system at run-time.

AURORA consists of three components: (1) a two-tiered mediation model and flexible data model support; (2) mediation methodologies and Mediator Author's Toolkits (MATs); and (3) query models and processing techniques.

The two-tiered mediation model mandates that data integration be performed in two steps: homogenization followed by integration. This model is designed to enable a divide-and-conquer approach towards data integration. Data sources are homogenized independently and in parallel, before they are integrated. AURORA provides specialized mediators to support homogenization and integration. The general principle of designing AURORA mediators is *"semi-automatic homogenization, automatic integration"*. The homogenization mediators are equipped with a Mediator Author's Toolkit (MAT) to assist mediator authors in working with semantics. A MAT mandates a mediation methodology that prescribes an approach of systematically identifying and resolving semantic mismatches. AURORA integration mediators provide a framework for automatic integration of homogenized sources.

AURORA provides mediators that support either the relational data model or an object-oriented data model (ODMG 2.0). The flexible data model support in AURORA allows applications to select data models according to their data access requirements. Moreover, sources can be integrated as long as they support a relational or an object-oriented interface (by themselves or through a wrapper). Thus, the amount of work in "upgrading" data model is reduced; this potentially allows a larger variety of data sources to be integrated with less effort.

Query processing in various AURORA mediators employs different techniques for manipulating data. In the homogenization mediators, query processing is based on Mediation Enabling Algebras (MEAs), which provide operators to enable manipulation of data to remove a wide range of se-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

mantic mismatches. These MEAs are also used for algebraic query optimization. The integration mediators deal with only instance level conflicts, since all other types of semantic differences have been removed by homogenization. AURORA integration mediators employ an integration operator that retains instance level conflicts and provide the applications with special query models, called Conflict Tolerant (CT) query models, to deal with the conflicts at query time. The key is to provide enough number of levels of tolerance without leaving the application overwhelmed.

Four types of AURORA mediators are described in this dissertation: AURORA-RH, AURORA-RI, AURORA-OH, and AURORA-OI. The homogenization mediators, AURORA-RH and AURORA-OH, are each equipped with a MAT. The CT query model used by the AURORA-RI mediator, and related query processing techniques, is described in detail. The relational mediators have been implemented. A prototype system that demonstrates the AURORA approach and techniques is also described.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# Acknowledgements

I thank my advisor, Professor M.Tamer Ozsu, for the directions and support he kindly and generously provided to me throughout the years. Tamer introduced me to the Center of Advanced Studies, IBM Toronto, and that opened the doors for my current position at IBM Almaden Research Center. After the arrival of my son in 1998, he has helped me through many difficult situations, both in my study and in other aspects of my life. I wish I could develop some strength in my character that is comparable to his. I consider myself fortunate to have Tamer as an advisor, in that if it is not for him, I would not be where I am today.

Special thanks go to Dr. Ling Liu, who was involved in my work for the first few years and provided interesting input into my project. I deeply admire Ling's drive and energy. Dr. Duane Szafron, who took time to look at part of my work, has helped me tremendously in improving the object-oriented framework of AURORA. My external examiner, Dr. Renee Miller, from University of Toronto, has raised interesting issues and kindly helped me to finish this thesis on time.

I thank my friend Vincent Oria for agreeing to do all the work for me when I repeatedly told him that I could never get that work done. Although I did finally do all the work myself, his words at the time were invaluable. I also thank Anne Nield for making all sorts of arrangement for me so I can work off campus. Anne also proof-read through my long long thesis, given that I wrote it, I know how much work that is. Thanks Anne!

My parents and my in-laws, Rudy and Heidi Kornelsen, have provided great support throughout my study. My mother came to Canada twice to take care of my young child so I can work on my project. Rudy and Heidi always opened their beautiful home to me, complete with nice meals and top-quality baby-sitting. I consider myself lucky to have wonderful parents and in-laws and my baby is lucky to have such wonderful grandparents, that he obviously loves very much.

Finally, I thank my husband Randal and my beautiful baby Murray, for their faith in me and for their support. In the first 6 months of his life, Murray exercised incredible amount of patience and attention span to allow me to work for a few hours a day while chewing up all his toys and learning all about space and computers (he also grew some eyelashes in his spare time!). When things get tough, Murray is always there to cheer me up. My dear husband Randal always had more faith in me than I myself do and he never allowed me to quit. Being with Randal has transformed my life for the better and the stronger. This achievement is so much yours as it is mine.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# Contents


**1 Introduction** 1
1.1 Motivation . . . 1
1.2 The AURORA Approach . . . 4
1.2.1 2-tiered Mediation Model and Flexible Data Model Support . . . 5
1.2.2 Mediator Author's Toolkit . . . 7
1.2.3 Query Models and Processing Techniques . . . 8
1.3 Organization of Dissertation . . . 10

**2 Related Work** 11
2.1 Classification of Approaches . . . 11
2.2 Federated Database Systems (FDB) . . . 13
2.2.1 Case Study: Pegasus . . . 14
2.2.2 Case Study: MSQL . . . 17
2.2.3 Case Study: MIND . . . 18
2.2.4 Modeling and Language Features in FDBs . . . 20
2.2.5 Schema and Query Translation Techniques in FDBs . . . 23
2.2.6 AURORA and FDBs . . . 23
2.3 Distributed Object Management . . . 24
2.4 The Mediator Systems . . . 26
2.4.1 TSIMMIS . . . 26
2.4.2 HERMES . . . 28
2.4.3 AURORA and Mediator Systems . . . 29
2.5 Paradigms For Dealing with Highly Dynamic Environments . . . 30
2.5.1 AURORA in Highly Dynamic Environments . . . 31
2.6 Query Processing and Optimization in IDAMSs . . . 31
2.6.1 New Enabling Operators and Evaluation Strategies . . . 33
2.6.2 Algebraic Transformation Rules . . . 34
2.6.3 Coping with Unavailable Optimization Information For Cost Modeling . . . 35
2.7 Summary . . . 37

**3 The AURORA Architecture** 38
3.1 AURORA Mediation Model . . . 38
3.1.1 Two-tiered Mediation Model . . . 39
3.1.2 Building A Data Mediation System with AURORA: A Scenario . . . 40
3.1.3 Two-tiered Model and Scalability . . . 40
3.1.4 Why Two-tiered? . . . 42
3.2 AURORA Architecture . . . 44
3.2.1 Data Sources and Wrappers . . . 44
3.2.2 AURORA Mediators As Middleware Components . . . 44
3.2.3 Mediator Author's Toolkits (MATs) in AURORA . . . 46
3.3 Semantics of Integrated Data Services . . . 47
3.4 Enabling Techniques in AURORA: A Roadmap . . . 50

**4 Relational Mediation Framework** 52
4.1 An Overview of the Relational Mediation Framework . . . 52
4.1.1 Service View . . . 53
4.1.2 Data Sources and Relational Wrappers . . . 53
4.1.3 Homogenization of Data Sources . . . 54
4.1.4 Integration of Homogenized Sources . . . 55
4.2 AURORA-RH Homogenization Framework . . . 56
4.2.1 The Homogenization Problem . . . 56


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.2.2 Mismatches and Resolutions . . . 57
4.2.3 AURORA-RH Primitives . . . 60
4.2.4 Homogenization Methodology and AURORA-RH . . . 62
4.2.5 Homogenization with AURORA-RH . . . 63
4.3 The Integration Framework and AURORA-RI . . . 72
4.3.1 Registrations . . . 72
4.3.2 Match Join . . . 73
4.3.3 An Integration Example . . . 74
4.4 Summary . . . 77

**5 Query Processing in Relational Mediators 78**
5.1 Conflict Tolerant Querying in AURORA-RI . . . 79
5.1.1 Motivation . . . 79
5.1.2 Instance Level Conflicts and Resolutions . . . 81
5.1.3 Conflict Tolerant Query Model . . . 84
5.2 Primitive Evaluation of Conflict Tolerant Queries . . . 86
5.3 Optimized Processing of Conflict Tolerant Queries . . . 89
5.3.1 CT Query Optimization: an Overview . . . 89
5.3.2 A Theory of Conflict Tolerant Query Processing . . . 90
5.3.3 Optimized Conflict Tolerant Query Evaluation Algorithms . . . 95
5.4 AURORA-RH Query Processing . . . 99
5.4.1 AQP Query Execution Engine and QEPs . . . 99
5.4.2 Query Rewriting . . . 99
5.4.3 AQP Query Optimization . . . 102
5.5 Related Work . . . 107
5.6 Summary . . . 110

**6 Object-Oriented Mediation Framework 113**
6.1 The Service View . . . 113
6.2 Data Sources and Wrappers . . . 114
6.3 Overview of the Homogenization Framework . . . 115
6.3.1 The Homogenization Scenario . . . 115
6.3.2 Homogenizing Views . . . 116
6.3.3 The Homogenization Facilities . . . 116
6.3.4 Homogenization Methodology . . . 118
6.3.5 AURORA-OH MEOs . . . 120
6.3.6 A Compact Homogenization Example . . . 122
6.3.7 MEOs and the Homogenization Methodology . . . 123
6.4 An Overview of the Object-Oriented Integration . . . 124
6.5 Basic Concepts in Object-Oriented Homogenization . . . 125
6.5.1 A Running Example . . . 125
6.5.2 A Framework for Describing Classes . . . 125
6.5.3 Navigation Methods . . . 130
6.6 Mediation Enabling Operators for Homogenization . . . 133
6.6.1 Regrouping MEOs . . . 133
6.6.2 The MEO for Object Generation: *OBJGEN* . . . 134
6.6.3 MEOs for Renaming and Deriving Methods . . . 136
6.7 Homogenization with AURORA-OH . . . 140
6.7.1 A Homogenization Methodology . . . 140
6.7.2 A Walk-Through of the Homogenization Example . . . 141
6.8 AURORA-OI: The Integration Mediator . . . 142
6.8.1 Oid Generation for Integrated Objects . . . 142
6.8.2 Fragments and Registrations . . . 143
6.8.3 Proxies . . . 145
6.8.4 Proxy Match Join: AURORA's Integration Operator . . . 147
6.9 Summary . . . 150

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

7 **Implementation of AURORA** 151
7.1 Choosing a Distributed Computing Framework . . . . . . 151
7.2 An overview of COM/DCOM and OLE-DB Technology . . . . . . 152
7.2.1 What is COM/DCOM? . . . . . . 152
7.2.2 The COM Way of Building Systems . . . . . . 153
7.2.3 OLE-DB Technology . . . . . . 154
7.3 AURORA Mediators as COM Components . . . . . . 157
7.3.1 Interfaces of AURORA Components . . . . . . 157
7.3.2 OLE-DB Providers as AURORA Wrappers . . . . . . 158
7.4 Implementation of AURORA-RH and AURORA-RI . . . . . . 159
7.4.1 Implementation of AURORA-RH . . . . . . 159
7.4.2 Implementation of AURORA-RI . . . . . . 167
7.5 Observations and Experiences . . . . . . 170

8 **Conclusions and Future Work** 171
8.1 Contributions . . . . . . 171
8.1.1 Mediation Model, Flexible Data Models, and Mediation Methodologies . . . . 171
8.1.2 Scalability and Flexibility of the AURORA Approach . . . . . . 172
8.1.3 Enabling Techniques . . . . . . 173
8.2 Experiences . . . . . . 173
8.3 Future Work . . . . . . 174

**Bibliography** 176

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# List of Tables

3.1 AURORA's Flexible Data Model Support . . . . . . . . . . . . . . . . . . . . . . . . 46

4.1 *categoryMap*: Domain Value Mapping for *Books.category* . . . . . . . . . . . . . . 71

5.1 Querying Integrated Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
5.2 Example Queries and Answers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
5.3 Example Queries and Answers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
5.4 Fragment Reduction with Selections . . . . . . . . . . . . . . . . . . . . . . . . . . 90
5.5 Transformation Rules for *pad, rename* and *deriveAttr* . . . . . . . . . . . . . . . . . 106

6.1 Example Global Schema and Source Homogenizing Views . . . . . . . . . . . . . . . 126
6.2 Source schemas and populations . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# List of Figures


1.1 The 2-tiered Mediation Model of AURORA . . . . . 6
1.2 AURORA Mediator Classification . . . . . 6

2.1 Schema Architecture of Federated Database . . . . . 14
2.2 Pegasus System Configuration . . . . . 15
2.3 Pegasus Functional Layers . . . . . 16
2.4 MIND Architecture . . . . . 19

3.1 Divide-and-conquer data integration of AURORA . . . . . 41
3.2 ISG Navigator as a Wrapper . . . . . 45
3.3 AURORA Application: uniform . . . . . 46
3.4 AURORA Application: mixed . . . . . 47
3.5 An AURORA Workbench . . . . . 48

4.1 Homogenizing View and Service View . . . . . 55
4.2 A Homogenization Example . . . . . 57
4.3 Source Tables . . . . . 58
4.4 Architecture of AURORA-RH Workbench . . . . . 64
4.5 $Books_p$: Result of RELmat . . . . . 65
4.6 $Sales_p$: Result of ATTRmat . . . . . 67
4.7 Derived population of relation *Books* . . . . . 71
4.8 Derived population of relation *BookSales* . . . . . 71
4.9 Relationship Between Homogenization and Integration Mediators . . . . . 73
4.10 Registered Fragments for Global Relation Books . . . . . 75
4.11 Derived Population of Global Relation *Books* . . . . . 76

5.1 *RAC* ( *Books*, *year*:MIN, *oprahClub*:ANY, *bestSeller*:DISCARD) . . . . . 83
5.2 *RTC(Books*, ANY) . . . . . 84
5.3 Compute CSET and content of $R'$ when $c_1$ = RandomEvidence or HighConfidence . . . . . 96
5.4 Compute CSET Phase for PossibleAtAll . . . . . 97
5.5 Value of $\delta_i$'s with $d$ = DISCARD . . . . . 98
5.6 Query Results . . . . . 98
5.7 Query Rewriting: $Q_1$ . . . . . 101
5.8 Query Rewriting: $Q_2$ . . . . . 102
5.9 Query Rewriting: $Q_3$ . . . . . 103
5.10 Query Rewriting: $Q_4$ . . . . . 104
5.11 Query Rewriting: $Q_5$ . . . . . 105
5.12 Query optimization example: first modification . . . . . 107
5.13 Query optimization example: second modification . . . . . 108
5.14 Query optimization example: third modification . . . . . 109
5.15 Query optimization example: fourth modification . . . . . 110
5.16 Query optimization example: fifth modification . . . . . 111
5.17 Query optimization example: sixth modification . . . . . 112

6.1 A Homogenization Methodology . . . . . 119
6.2 Logical and Implementation Schemas of Class Professor . . . . . 129
6.3 Use of Proxy for Uniform Access . . . . . 145

7.1 Use of OLE-DB Interfaces and the Role of Rowsets . . . . . 156
7.2 AURORA Components as COM Components . . . . . 158
7.3 The Making of AURORA Wrappers Using OLE-DB Providers . . . . . 159


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

7.4 Implementation of AURORA-R Mediators . . . . . . . . . . . . . . . . . . . . . . . 160
7.5 Main Window of MAT-RH GUI . . . . . . . . . . . . . . . . . . . . . . . . . . . . 161
7.6 Import pop-up menu . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 162
7.7 SME-1 pop-up menu . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 163
7.8 Dialog Window for the *RELmat* Transformation . . . . . . . . . . . . . . . . . . . . 164
7.9 Dialog Window for the *ATTRmat* Transformation . . . . . . . . . . . . . . . . . . . 165
7.10 Pop-up Menu of RLE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 166
7.11 First Dialog for Specifying Domain Structural Functions . . . . . . . . . . . . . . . . 167
7.12 Second Dialog for Specifying Domain Structural Functions . . . . . . . . . . . . . . . 168
7.13 First Dialog for Registering Relations with Integration Mediator . . . . . . . . . . . 169
7.14 Second Dialog for Registering Relations with Integration Mediator . . . . . . . . . . 170

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# Chapter 1

# Introduction

A significant challenge facing the database field has been accessing multiple, heterogeneous data sources. Connecting to multiple data sources from a single application is easy, but accessing them transparently is difficult due to the heterogeneities in platform, structure, and semantics among the sources. An *Integrated Data Access Middleware System* (IDAMS) is a software system that provides applications with one-stop data services using data from multiple heterogeneous and autonomous data sources. IDAMSs are middleware systems; they are non-intrusive, respecting the autonomy of the data sources involved, and they deal with the complexities of accessing data from multiple sources on the applications' behalf. IDAMSs support *service views*, which are interfaces through which data from multiple sources can be accessed transparently. A service view may consist of a loose collection of source schemas presented in a common data model acceptable to the applications. In this case, the applications pose queries against multiple schemas that often overlap in semantics; mismatches and inconsistencies among these schemas must be dealt with at query time. More often, a service view is *integrated*, allowing applications to access multiple sources as if they are a single source. Research on IDAMSs dates back to the 1980s and has seen substantial progress [47, 7, 51, 51, 5, 44, 6, 88, 12, 50]. Recently, this area of research is experiencing a resurgence due to the advances in distributed computing technology and the fast growing availability of the Internet. These advances give rise to new application scenarios and pose new requirements on IDAMSs. This dissertation describes a project, AURORA [97, 94, 98, 95], that develops frameworks and techniques to address these challenges.

## 1.1 Motivation

The availability of the Internet and the Web changes the way people use digital information; it gives rise to new applications, such as electronic commerce and digital libraries, that use the Web as a media for conducting business and exchanging information anytime anywhere. These changes bring the following new dimensions (among others) into IDAMSs:



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. **Large scale of access scope.** Highly distributed, on-line applications, such as electronic commerce and digital libraries, demand access to a large number of data sources around the globe for information gathering. This calls for a paradigm of *scalable mediation*, to allow large number of data sources to be integrated in a dynamic and incremental manner. Scalability in this context means the following:

   - Integration of a large number of sources should have the same degree of complexity and require the same level of expertise as integrating a small number of sources. For instance, integrating 500 sources should be as easy as integrating 2 data sources.
   - The effort involved in adding/removing a data source should be manageable. For instance, tools and methodologies should be provided to assist the users in specifying how source data should be transformed in order to be integrated.

2. **Highly dynamic and diverse environment.** Data sources come and go autonomously at unpredictable rates and vary in availability and capabilities. Applications consuming the data range from business applications that usually prefer the relational data model, to multimedia applications that work with object data. These characteristics of the environment calls for a paradigm of *flexible mediation*. Flexibility in this context means the following:

   - Flexible data model support. Applications should be given the flexibility of using a data model of their choice. The effort in upgrading data models of the sources for integration purpose should be reduced.
   - Dynamic access scope. Inclusion/exclusion of a data source from the access scope should have no impact on the availability of the service view or the participation of other data sources.

Existing IDAMSs are built with one of two paradigms: source-driven integration and application-driven integration.

#### Source-Driven Integration

In this paradigm, an integrated service view is derived by resolving the various types of heterogeneities among the participating sources. Such a derivation is defined by an *integration specification*, which references the structure and semantics of individual sources directly. Adding or removing a source requires modification to this specification and/or to the service view itself. When the number of sources is large and the sources have unpredictable availability, the integration specification becomes difficult to maintain. Federated database systems [47, 88, 5, 44, 12] and some mediator systems [74] use this paradigm.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**Application-Driven Integration**

In the application-driven integration paradigm, the service view is defined based on the application's view of data, regardless of the structure or semantics of the participating sources; it is a contract of data service between the application and the IDAMS. For a source to participate in a service view, the IDAMS must be able to "understand" the data provided by this source. Representative systems using this paradigm are SIMS [6] and Information Manifold (IM) [50]. In SIMS, the service view is a knowledge base that can be queried in the target applications' terms. A participating source describes its content in a *domain model* consisting of definitions of all the relevant terms and values in a given ontology. SIMS then uses this knowledge for source selection and query planning when processing queries. In IM, the service view is a relational view and a participating source describes its content as a materialized view of the service view. In both SIMS and IM, sources join and leave the service view autonomously without impacting on the availability of the service view or the participation of other sources.

The application-driven integration paradigm does not remove any semantic or structural heterogeneities among the data sources but it allows for a divide-and-conquer approach towards data integration: rather than examining a large number of sources and trying to piece them together to form a service view, each source can be "hooked" into the service view independently. This effectively decomposes the task of "integrating a large number of sources" into a set of smaller tasks of "hooking individual sources" into a common service view. Conceivably, each source can be worked on independently and in parallel. While this is a promising idea, various issues must be resolved to make such a system practical. This dissertation addresses some of these issues, including the following:

**Easy participation of data sources.** Making the source data understandable in the context of a canonical application model (the service view) requires removal of a wide range of mismatches in structure and semantics between the source schema and the service view. As demonstrated by previous work [7, 85, 44, 41, 64], working with semantics is hard. Therefore, provision of tools and facilities in support of this task is an important factor in usability of an IDAMS. IM does not provide such facilities. SIMS provides a model building tool, but describing source data in a given ontology requires a significant level of expertise.

**The impact of data integration on query processing must be taken into account.** Although much is known about classifying and resolving semantic heterogeneities [44], the impact of this process on query processing is seldom discussed. Most previous approaches assume that an IDAMS uses the same set of data manipulation operators as traditional DBMSs, although these operators must be evaluated with different techniques since the operand data may reside in heterogeneous and autonomous sources. In order to entertain queries against the service views, an IDAMS needs data manipulation operators that are unknown to traditional DBMSs. For instance, in addition to select, join, and project, it may be necessary to apply functions to columns in a table,



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to transform table names into data values, and so on. It is important to define these operators, and to incorporate them into an algebraic framework which enables algebraic or cost-based query optimization in an IDAMS. Without this framework, service view queries may become extremely expensive to evaluate in the presence of information overload.

**Instance level conflicts must be dealt with.** Instance level conflicts arise when sources provide inconsistent data on the same application entity. Most systems dealing with large scale integration, such as IM, do not deal with these conflicts. The reason may be that, to protect the applications from the scale of the access scope, an IDAMS aims at creating a single-source illusion, limiting treatment of instance level conflicts to detecting and resolving them before query evaluation. Previous work has demonstrated that conflict detection and resolution is fundamentally expensive, requiring retrieval and manipulation of large amounts of source data, and the situation gets worse when the number of sources grows. However, it is unrealistic to assume full consistency among a large number of autonomous sources; instance level conflicts must be handled in spite of the scale of the integration. Since creating single-source illusion significantly limits conflict-handling options, a compromise approach is established in AURORA by exposing the conflicts to the applications in a controlled and easy-to-manage manner, and in coarse granularities. Such a technology will improve the practical value of IDAMSs.

In this dissertation, the above-described issues are addressed within the context of the AURORA project. AURORA is based on a 2-tiered mediation model that realizes the application-driven integration paradigm. The goal of AURORA is to enable *scalable mediation*, so that adding and removing data sources is easy, and *efficient mediation*, so that mediator queries can be processed without retrieving irrelevant source data. Several enabling techniques are developed:

1. The *Mediator Author's Toolkits* that assist in the tasks of making sources "understandable" in the context of a canonical application model (the service view). These tools allow easy participation of data sources in the access scope of a target service view.

2. The *Mediation Enabling Algebras* that are specially designed to support mediator query processing and optimization.

3. *Conflict tolerant query model* and processing techniques for querying potentially inconsistent data.

## 1.2 The AURORA Approach

The AURORA project consists of three components: (1) 2-tiered mediation model and flexible data model support; (2) mediation methodologies and Mediator Author's Toolkits (MATs); and (3) query models and processing techniques. The general architecture adopted by AURORA is the mediator architecture [91]. AURORA mediators cooperate with one another to achieve data integration; they can be composed. However, AURORA mediators perform specific types of mediation prescribed by



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the AURORA *mediation model* that defines the tasks involved in the data integration process, and the relationships among them.

### 1.2.1 2-tiered Mediation Model and Flexible Data Model Support

The goal of mediation in AURORA is to support data access through service views. A service view is a database schema designed to satisfy the data access requirements of a class of applications. For these applications, the service view is a schema designed according to their view of data in a data model they are familiar with. Given a service view, AURORA must enable the relevant applications to access data residing in multiple sources through this service view, typically by entertaining queries posed against it. AURORA achieves this goal by first establishing a mediation model, and then developing techniques to achieve each task prescribed by this model.

AURORA models data integration as a 2-step process: *homogenization* followed by *integration*; each step is performed by respective mediators (Figure 1.1). Each integration mediator supports a pre-defined service view. Data sources can participate in one or more of these service views by contributing data towards them. To do this, they must first be *wrapped* and then *homogenized* against the service view(s) they want to participate in. Wrappers provide conformity in data model and query languages of data sources; they do not deal with any other types of heterogeneities. Homogenization removes idiosyncrasies of a data source so that it conforms to the target service view in structure and semantics. If a source participates in more than one service view, it must be homogenized multiple time - once against each target service view - but it needs to be wrapped only once. Homogenization is where individual sources "adapt" themselves into a form that is ready to be included in the access scope of the target service view; homogenization mediators can be thought of as *data adaptors*. The result of homogenization is a *homogenizing view* that satisfies a few conditions, prescribed by the AURORA's mediation model, that ensure the data provided through this view be interpreted appropriately by the target service view. Homogenization involves only one data source; multiple sources can be homogenized independently and in parallel. After being homogenized, a data source participates in the target service view by describing the homogenizing view it supports to the relevant integration mediator. A data source can remove itself from the scope of a service view by informing the relevant integration mediator that it contributes data towards this service view.

An integration mediator is responsible for providing data through a pre-defined service view using data contributed by participating sources through respective homogenizing views. Since all the sources are homogenized before participating in the service view, integration is fully automatic. In comparison, homogenization is a more difficult task and is assisted by AURORA tools.

AURORA's mediation model realizes the application-driven integration paradigm. It defines a divide-and-conquer approach towards data integration. In particular, it enables decomposition of the data integration task into two smaller and simpler tasks: homogenization and integration. While



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

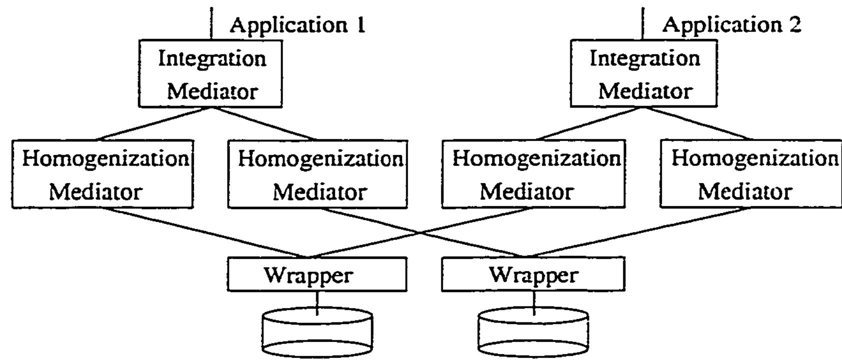



Figure 1.1: The 2-tiered Mediation Model of AURORA

integration is automatic, homogenization requires human interference and it can be performed on individual sources independently and in parallel. In this fashion, multiple parties can independently contribute to the same integration effort. As discussed later, in Section 3.1.4, the mediation model also facilitates the decomposition of complex technical issues in large-scale data integration systems. such as query processing, into more manageable, simpler problems.

Traditionally, an IDAMS supports a *canonical data model* (CDM) as the only data model via which data are accessed. The CDM is often chosen for its power of expressing resolutions of mismatches and for representing a large variety of data types. In recent years, object models have been favored as the CDM by a number of systems [5, 89, 24, 44, 12]. However, based on the data access requirements and other historical or practical factors, applications may prefer to use a certain data model. For example, the majority of existing database applications use a relational data model. Forcing these applications to use an object data model may reduce the practical appeal of an IDAMS. Furthermore, using an object model as the CDM may introduce the complexities of object-oriented DBMSs even when it is not called for.

The AURORA architecture and framework are designed to allow applications to select a data model that best satisfies their data access requirements. AURORA supports two popular data models: relational and object-oriented. Each AURORA mediator is characterized by the type of mediation it performs - homogenization or integration - and the canonical data model it supports - relational or object-oriented. Figure 1.2 shows the various AURORA mediators. Necessary guidelines and techniques are provided to allow these mediators to co-exist and cooperate, as described in Chapter 3.

| Canonical Data Model / Mediator Type | Relational | Object-Oriented |
|---|---|---|
| Homogenization | AURORA-RH | AURORA-OH |
| Integration | AURORA-RI | AURORA-OI |

Figure 1.2: AURORA Mediator Classification



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A general principle of designing AURORA mediators is *"semi-automatic homogenization, automatic integration"*. The activity of homogenization deals with a wide range of semantic and structural mismatches between a source schema and a service view. This is a difficult task since it requires dealing with semantics. AURORA homogenization mediators are equipped with a *Mediator Author's Toolkit* (MAT), which provides guidelines and facilities to a *mediator author* for performing homogenization. MATs are further described in Section 1.2.2. AURORA integration mediators deal with a small number of conflicts and are fully automatic; they have no MAT attached. All four types of AURORA mediators have been designed and the implementation of two of them, AURORA-RH and AURORA-RI, have been completed. Implementation of AURORA-OH and AURORA-OI mediators is on-going and is beyond the scope of this dissertation. AURORA mediators are described in Chapters 4 to 6.

### 1.2.2 Mediator Author's Toolkit

*Instance level conflicts* arise when different sources record conflicting values about the same application entity. *Semantic and structural mismatches* arise when sources model the same application domain differently. Instance level conflicts are not handled until multiple sources "meet" at an integration mediator. All structural and semantic heterogeneities between a participating source and the target service view must be resolved during homogenization. Previous research gives rise to two approaches for dealing with structural and semantic mismatches:

1. Automatic, by using a knowledge base and/or ontology. The structure, semantics, and content of a participating source must be incorporated into a knowledge base or described using an ontology. The IDAMS is then responsible for automatic integration of all participating sources by inferring relevance and correspondence among the source data using the knowledge base. SIMS [6] takes this approach.

2. Manual, by means of a human using the provided language constructs to resolve mismatches. Significant progress has been made in classifying the types of semantic and schematic mismatches and in resolving them [44].

Describing the meaning of a source schema using a knowledge representation language or a given ontology requires a significant level of expertise, making the automatic resolution approach difficult to deploy. Comparing a source schema and a service view to identify all mismatches between them and resolving these mismatches may overwhelm a human when the schemas are large, and when a large variety of mismatches are present between the two schemas.

In AURORA, structural and semantic mismatches are dealt with by *mediator authors*, individuals who have good knowledge of both the source schema and the target service view. However, the mediator authors are provided with a GUI-driven *Mediator Author's Toolkit* (MAT) to help them to work with semantics. A MAT consists of two parts: a mediation methodology and a set of



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

transformations that allow expression of resolutions of various mismatches. The mediation methodology guides the mediator author to identify and resolve mismatches in well-defined steps; in each step, transformations specially designed for resolving certain types of mismatches can be used to express the resolutions chosen by the mediator author. Intuitively, a MAT provides a skeleton for constructing a mediator; the mediator authors must fill various parts of this skeleton using their understanding of the mismatches and the resolutions of their choice. The transformations they choose are represented internally as expressions in *Mediation Enabling Algebras* (MEAs), algebra systems that specialize in manipulating heterogeneous data, as described below.

### 1.2.3 Query Models and Processing Techniques

AURORA mediators entertain application queries and hence provide data services: they do so by sending queries to relevant AURORA components, such as mediators and wrappers, and assemble answers to the application queries based on the results returned by these components. Query processing in different types of mediators requires different techniques to be developed.

**Query Processing in Homogenization Mediators**

AURORA homogenization mediators must be able to answer queries posed against the homogenizing views - views generated by the homogenization process. Although much is known about classifying and resolving semantic heterogeneities [44, 64] that may be encountered during homogenization, the impact of these resolutions on mediator query processing is seldom discussed. AURORA homogenization mediators provide *homogenization operators*. These operators are specially designed for transforming data during homogenization and they form algebras, called *Mediation Enabling Algebras* (MEAs), that are suitable for query optimization and processing in homogenization mediators. MEAs allow the impact of data integration on query processing to be identified and taken into consideration. Work in this category includes the following:

1. Development of MEAs for each AURORA homogenization mediator.

2. Development of transformation rules for each MEA to facilitate algebraic query optimization techniques for the corresponding mediators.

Homogenizing views as well as queries against them are expressed in MEAs. The view expressions are used to modify a view query. The modified expression is then manipulated by an algebraic query optimizer that pushes, whenever possible, the operations into the underlying data source so as to cut down the volume of data fetched into the mediator. Data returned from the underlying sources are further processed by the mediator to produce query results. In this process, MEA operators are used to restructure, transform, and assemble data.

**Query Model and Processing in Integration Mediators.**



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Query processing in integration mediators not only requires MEAs to be developed, as in homogenization mediators, it also requires techniques for dealing with instance level conflicts. The approach employed by AURORA is to expose these conflicts to the applications at query model level.

Since integration mediators deal with homogenized sources, the only operator needed is an integration operator that matches and combines the homogenized data pieces provided by participating sources, to produce integrated data suitable to be served to the applications. Both integration mediators in AURORA, AURORA-RI and AURORA-OI, define such an integration operator.

Traditionally, an IDAMS attempts to create a single-source illusion, that is, it allows the applications to access the multi-source data as if they reside in a single-source, with no inconsistencies. To follow this tradition, AURORA integration mediators would have to detect and resolve all instance level conflicts without impacting on the query model. This is not a desirable approach since it can be expensive and, fundamentally, little can be done to cut the cost [21, 15]. AURORA integration mediators employ a new approach towards instance level conflict handling, called *conflict tolerant querying*. In this approach, instance level conflicts are not resolved at schema integration time, rather, they are exposed to the applications, which deal with them using a conflict tolerant query model. This query model defines query semantics based on possibly inconsistent data; conflicts are tolerated to a few levels to be specified by the users at query time, and conflict resolutions are only performed to produce conflict-free results. The key is to keep the query tolerance levels simple for the applications to understand and to use.

A conflict tolerant query model, a CT query model, and related processing techniques have been developed for AURORA-RI, the relational integration mediator. This query model currently supports three levels of conflict tolerance. With this model, it is possible to reduce the overhead of conflict detection and resolution and to develop new techniques to optimize query processing. Fundamentally, the CT-query approach allows applications and the mediation systems to handle conflicts at a coarse granularity and achieve better query performance when conflict resolution requirements are relaxed and/or data contain occasional conflicts. CT query model and processing techniques for the AURORA-OI mediator constitute a future research topic and are beyond the scope of this dissertation.

It is conceivable that the CT-query model gives rise to new data manipulation operators that, together with the data integration operator, form an algebraic framework that can be used as the basis for optimized processing of CT-queries in an integration mediator. However, the current work on CT-queries is not yet in this stage. Rather, the current work focuses on establishing the query model itself and developing optimization strategies. The developed strategies are presented as query optimization algorithms, rather than as algebraic transformation rules. Using formal MEAs to optimize CT-query processing in AURORA integration mediators is an issue for further research.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

## 1.3 Organization of Dissertation

This dissertation is organized as follows. Chapter 2 contains a review of previous work related to AURORA. Chapter 3 contains a general description of AURORA's architecture and a road-map to the techniques developed. Chapter 4 describes the homogenization and integration frameworks in the relational context. Chapter 5 describes the query processing frameworks and techniques developed for the relational mediators. Chapter 6 describes the homogenization and integration framework in the object-oriented context ( a query processing framework and techniques suite in the object-oriented context similar to those described in Chapter 5 is beyond the scope of this dissertation). Chapter 7 describes the current implementation of the prototype system. Chapter 8 contains conclusions and future work.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# Chapter 2

# Related Work

In Chapter 1, a distinction is made between two paradigms of data integration performed by an IDAMS: source-driven and application-driven. This is a high level distinction based on "what" an IDAMS does. This chapter reviews "how" previous IDAMSs work. Even though many previous systems perform source-driven integration, the architecture, model, languages, and query processing techniques developed are relevant to AURORA. In this chapter, these works are reviewed and AURORA is positioned with respect to them. More specific comparisons between techniques employed by AURORA and previous work are included in later chapters when these techniques are described in detail.

## 2.1 Classification of Approaches

IDAMSs facilitate data access through a service view, based on data contributed by a set of heterogeneous sources. They must have knowledge as to how the service view is to be derived from the sources. This knowledge can be used to load data into the IDAMS if the supported service view is to be materialized. If the supported service view is virtual, the IDAMS can use this knowledge to process queries against the service view by decomposing these queries into subqueries, sending the subqueries to various sources for execution, and using the returned data to assemble the query answer. Based on the representation, acquisition, and use of this knowledge, previous approaches can be classified into three categories:

1. *Procedural integration.* The knowledge is provided to the IDAMS as a derivation specification constructed by a mediator author, who identifies and resolves all the structural and semantic mismatches among the participating sources, and specifies how the service view is derived. The system uses an underlying algebraic or logical framework to "execute" the integration specification in order to derive view data. Systems in this category include Multibase [47], Mermaid [88], MRSDM [51], Omnibase [82], Pegasus [5], UniSQL/M [44], MIND [24], HERMES [87], TSIMMIS [74], Garlic [12], IRO-DB [30], and many others.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. *Intelligent integration.* The knowledge is provided to the IDAMS as a canonical model of the application semantics in the form of a knowledge base or an ontology. Each participating source provides descriptions of the data it provides in terms of this canonical semantic model: these descriptions are incorporated as part of the knowledge base. The IDAMS reasons about the semantics of source data to determine how to match and merge them to derive data described by the service view. Systems in this category include Carnot [17], SIMS [6], InfoSleuth [8], the Context Interchange approach [31], DIOM [53], and others.

3. *Declarative integration.* The knowledge is provided to the IDAMS as a collection of source descriptions, each specifying the relationship of a source schema to the service view. For instance, a source schema could be described as a materialized view of the service view. As another example, a source schema could be described as containing a collection of objects which is a sub-extent of a class in the service view. The IDAMS does not reason about semantics, but uses a mechanism that interprets the relationships of various sources to the service view and "pieces" the source data together to produce view data. Systems in this category include Information Manifold [50], DISCO [89], and others.

In the intelligent integration approach, adding a source into the integration scope means "hooking" its description onto the underlying semantic model, without impacting on the service view itself, or the participation of other sources. The drawback of this approach is that it requires significant levels of expertise to construct source descriptions, which may make hooking a new source into the service view a difficult task. If so, the integration process is not scalable since adding a source is difficult. It is also not clear how this approach facilitates the processing of complex queries such as those involving the use of aggregation functions and nested queries. Procedural integration systems often do not facilitate scalable construction of the integration specification, making this paradigm weak in its support for scalability of integration. Declarative integration systems are scalable, since each data source can be described in regard to the service view independently. However, these systems often provide fewer facilities for dealing with various mismatches between the sources and the service view. Such facilities may require a rich set of constructs for describing a source schema in terms of a given service view. The presence of these constructs may make it difficult, if not impossible, to establish an algorithm for piecing together source data to produce view data.

AURORA retains the scalability of the declarative integration approach, while enhancing its facilities for dealing with heterogeneities by incorporating features of the procedural integration approach - namely, "procedural homogenization, declarative integration". Once the heterogeneities are identified, resolving them is relatively easy [42, 7]. The most difficult part in dealing with heterogeneities is in identifying them in the first place, since this requires understanding, comparing and matching of the "meaning of things". Intelligent integration systems reason about semantics based on source descriptions in order to identify and resolve heterogeneities automatically. Other systems leave this task to a human, but provide facilities for resolving them once they are identified;



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

AURORA falls into this category. However, AURORA differs from existing systems in that it provides *mediation methodologies* to assist the mediator authors in systematic identification and treatment of semantic and structural heterogeneities. As such, work on reasoning about semantics is less relevant to AURORA than work on providing facilities for resolving heterogeneities. However, for completeness of this survey, a brief review of the work done on reasoning about semantics is given below.

**Reasoning About Semantics.** A good overview of various techniques for reasoning about the meaning and resemblance of heterogeneous objects is given in [36]. In [64] and [63], the theory of information capacity equivalence and dominance is used to develop tests that can be used to check for correctness and other properties of semantic transformations. Intelligent integration systems such as Carnot [17], SIMS [6], and InfoSleuth [8] use a knowledge base and/or ontology to "understand" the semantics of various sources, to reason about it, and eventually, to integrate the sources based on such understandings. Work in this category also includes context mediation [84], context interchange [31], the query mediation approach [77], dynamic query routing for a digital library application [54], query reformulation using semantic knowledge represented by integrity assertions and mapping rules [28, 29], and others.

The rest of this chapter reviews the architecture, model, and language features developed in procedural and declarative IDAMSs. These systems are developed in four areas of research: federated database systems, distributed object management systems, mediator systems, and systems that deal with dynamic integrations. Mediator query processing techniques are discussed separately in Section 2.6.

## 2.2 Federated Database Systems (FDB)

Federated database systems (FDBs) [47, 16, 5, 44, 88, 12, 24] represent a traditional paradigm of building IDAMSs. FDBs perform procedural integration. Compared with declarative integration systems, FDBs are characterized by the use of a monolithic integration specification that resolves a wide range of semantic heterogeneities among a small number of participating databases.

This section contains case studies of a few representative systems, and a review of various areas of research in the context of federated database systems. The case studies focus on the paradigm and architecture of the systems. Specific technical issues are reviewed by category.

Traditionally, federated database systems are built based on a five-level extended schema architecture, as shown in Figure 2.1 [85]. A *local schema* is the conceptual schema of a data source, referred to as a *component database*; it is expressed in the data model of the component database. Hence different local schemas may be expressed in different data models. A *component schema* is derived by translating a local schema into a *canonical data model*. An *export schema* is a subset of a component schema that is made available to the federation. A *federated schema* is an integration of multiple export schemas. The *external schemas* are the views exposed to applications.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

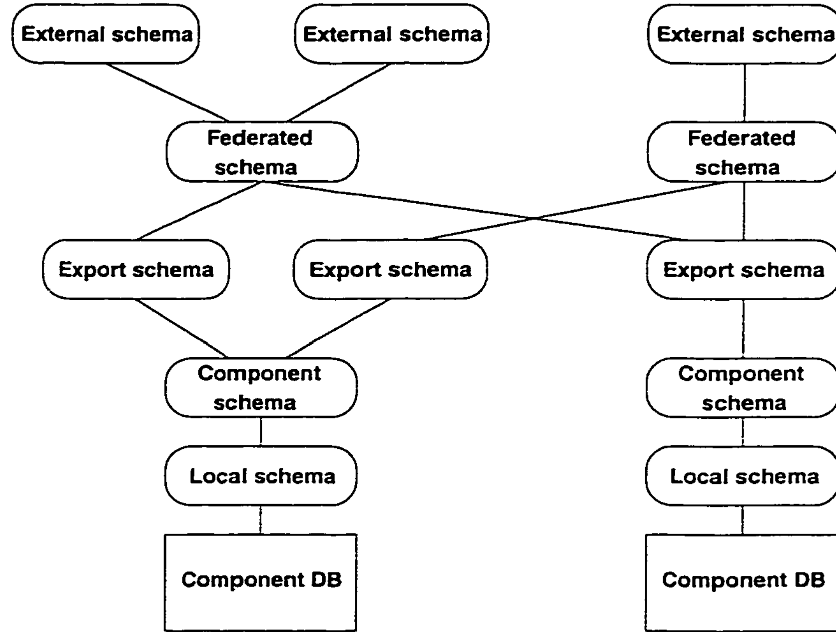



Figure 2.1: Schema Architecture of Federated Database

Based on whether the federated schema is a loose collection of imported schemas represented uniformly in a common data model or an integrated schema, called a *global schema*, the federated database systems are classified into two categories: *tightly-coupled* and *loosely-coupled*. Tightly-coupled systems support a global schema via which queries against data in the federation can be processed. Construction of this global schema requires that the semantic heterogeneities among participating databases be resolved. Users who query the global schema are presented with a single-source illusion. In loosely-coupled systems, the federated schema is a collection of possibly inconsistent schemas represented uniformly in a common data model. The users query this collection of schemas using a *multidatabase query language*. Since the schemas are not integrated, heterogeneities among them must be resolved by the users at query time. The distinction between tightly-coupled and loosely-coupled systems is blurred in systems such as Pegasus and UniSQL/M. In these systems, schema integration is optional. Users can choose to perform integration to various degrees and resolve other semantic discrepancies at query time.

### 2.2.1 Case Study: Pegasus

The Pegasus project [5] is built around the architecture given in Figure 2.1, but is not as elaborate. The schema architecture of Pegasus is given in Figure 2.2 [5]. Implementation of this architecture is given in Figure 2.3 [5].

Pegasus [5] uses an object-oriented, functional data model, Iris, as a framework for uniform interoperation of multiple heterogeneous databases. The unifying data definition and manipulation



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

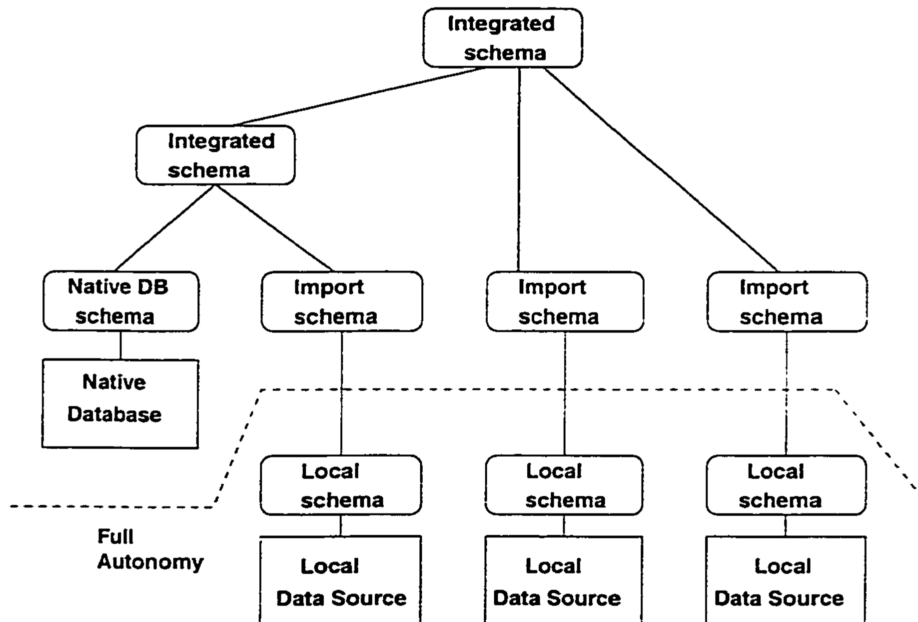



Figure 2.2: Pegasus System Configuration

language is the Heterogeneous Object Structured Query Language, the HOSQL, which provides statements to unify, manipulate, and query multiple heterogeneous databases.

A component database, referred to as a *local database*, must be *registered* with Pegasus and its schema must be *imported*. Registration describes the types in the local DBMS, the network protocols, network nodes, machine types, and so on. For each type of local database, Pegasus provides a module, called *local translator*, that maps a local database schema into the Pegasus data model and also translates queries expressed in HOSQL over this schema into local query language (such as relational SQL). These modules are used to import local database schemas.

After being imported, multiple data sources can interoperate through Pegasus in that HOSQL can be used to query the union of the imported schemas. Construction of integrated schemas from multiple imported schemas is optional and deals with semantic and schematic heterogeneities among the imported schemas. This integration is supported by specially designed HOSQL language constructs. The major constructs are (1) creating supertypes of types defined in the underlying database; (2) creating *derived functions*; and (3) creating *foreign functions*. These constructs are illustrated in the following example:

**Example 2.2.1** [HOSQL for schema integration]
Suppose there are two imported types, Student1 and Student2, with functions Grade and Points defined on them, respectively. Also assume that the two functions use different grading systems. For instance, function Grade might return a value in {A, B, C, D, E}, while function Points returns an integer value between 1 and 10. The user can define two functions, Map1 and Map2, to convert



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

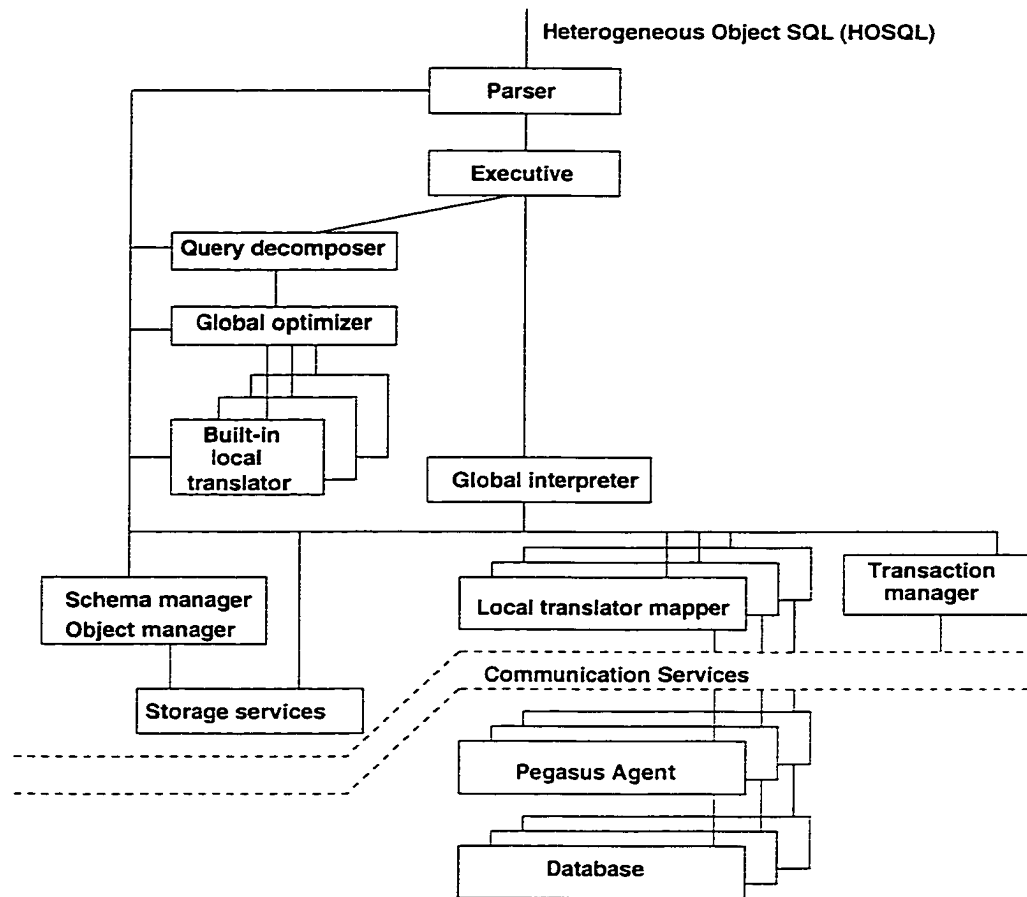



Figure 2.3: Pegasus Functional Layers

Points and Grade to a common grading system. The integrated type Student can then be defined as follows:

> **Create Supertype** Student **of** Student1, Student2;
>
> **Create Function** Score(Student x) $\rightarrow$ **REAL** r **AS**
>
> > **IF** Student1(x) **THEN** MAP1(Grade(x))
> >
> > **ELSE IF** (Student2(x) **THEN** MAP2(Points(x))
> >
> > **ELSE ERROR**

Type STUDENT is created as a supertype of imported types Student1 and Student2. The function SCORE is a derived function. Functions MAP1 and MAP2 are foreign functions. □

As shown in Figure 2.3, a HOSQL query is decomposed into an operation tree whose operators are commands for performing global joins, to pass parameters, and to synchronize execution of parallel steps. The leaf nodes of this tree are queries against local databases. These queries are



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sent to the *local translator* module to be translated into the query language supported by the local database. These translated queries are then sent to the *local translator mappers* that perform system level mappings on the query before passing it to the *Pegasus agent* that runs on top of the local databases. The local translator mapper and the Pegasus agent work together to submit the query for execution and to collect the query result back into Pegasus. The query processing activities, as well as global transactions, are controlled and synchronized by the *global interpreter*.

### 2.2.2 Case Study: MSQL

MSQL [51, 52, 34, 65, 86, 66] is designed as an extension of SQL for querying multidatabase systems. In MSQL, multiple databases are visible to the users who can refer to attributes, tables, and databases. However, the users must also be responsible for the consistency of the data retrieved. Location of data sources is made transparent to the users in that site-dependent access protocols are transparent. The most important extension from SQL to MSQL is the notion of *multiple queries*, which enables expression of multiple queries to several (related) databases in a single query. Central to the multiple query facility are the concepts of *multiple identifiers* and *semantic variables*. A multiple identifier is a means to refer to multiple relations/attributes using a single identifier, as illustrated in Example 2.2.2. A semantic variable is a variable that ranges over multiple databases, relations, or attributes, as illustrated in Example 2.2.3. These two concepts allow factorization of single multidatabase queries into a set of elementary queries against individual databases. The main features of MSQL are illustrated by the following examples.

**Example 2.2.2** [Multiple identifiers in MSQL]
Assume there are three bank databases, $B_1$, $B_2$ and $B_3$, each containing a relation *client*. A *multi-database* $BANKS$ can be created as follows:

**CREATE MULTIDATABASE** $BANKS(B_1, B_2, B_3)$

To retrieve client information from all three databases, the following MSQL statement can be used:

**USE** $BANKS$
**SELECT** *
**FROM** *client*

In this query, *client* is a multiple identifier. At query processing time, this query is replaced by three queries retrieving client information from the three databases, respectively. □

**Example 2.2.3** [Semantic variables in MSQL]
Continue with Example 2.2.2, assume all three databases contain a relation that describes branches,



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

but in $B_1$, this relation is named *branch*, in $B_2$, $BR$, and in $B_3$, $BRCH$. The following MSQL query retrieves all the bank branches on 101 Street:

**USE** $BANKS$
**LET** $x$ **BE** *branch*, $BR$, $BRCH$
**SELECT** *
**FROM** $x$
**WHERE** street = "101"

In the above query, $x$ is a semantic variable. At query processing time, the above query is replaced by three queries produced by replacing $x$ with *branch*, $BR$, and $BRCH$ respectively. The use of semantic variables enables expression of three queries against three databases in a single, compact MSQL query. □

As one can infer from the above examples, to use the original MSQL correctly, the user must have good knowledge about the scope of databases and the data they hold. In some later work [65, 66], MSQL is extended to allow external functions, class attributes, implicit joins and type casting. These are language constructs for users to resolve semantic mismatches among the imported schemas concisely at query time. The query processor for this extended variety of MSQL must be "intelligent" enough to expand, factorize and decompose queries.

### 2.2.3 Case Study: MIND

The METU INteroperable DBMS (MIND) [24] is an implemented FDB system that supports integrated access to multiple heterogeneous and autonomous databases. MIND is able to access Oracle 7, Sybase, Adabas and MOOD - an object-oriented DBMS developed by the same group. Like many other FDB systems, MIND is built around the 5-layer schema architecture, as shown in Figure 2.1. The canonical data model and query language of MIND are object-oriented. MIND differs from other FDB systems in that it uses CORBA as the infrastructure for managing the distribution and system level heterogeneities. As such, the system has a distributed and object-oriented architecture, as shown in Figure 2.4. All components in this diagram are built as objects that communicate with one another via an object request broker.

The central components of MIND are two object classes: the Global Database Agent (GDA) class and the Local Database Agent (LDA) class. Objects of these classes can be created by an object factory. These objects are described in terms of their functionalities as follows:

1. A LDA object is responsible for the following:

    - Maintaining export schemas provided by the local DBMSs. This schema is represented in the canonical data model.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

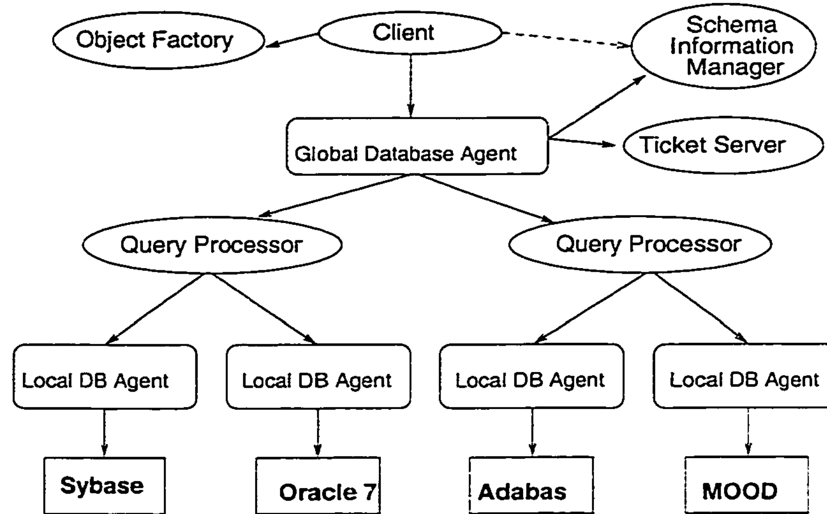



Figure 2.4: MIND Architecture

- Translating queries received in the canonical query language to the query language of the local DBMSs.
- Submitting queries to local DBMS for execution and collecting the query results.

2. A GDA object is responsible for:

- Parsing, decomposing, and optimizing the queries according to the information obtained from the Schema Information Manager object.
- Global transaction management that ensures serializability of multidatabase transactions without violating the autonomy of local databases.

When a user wants to query MIND, a GDA object is created by the object factory. The location and implementation transparency for this object is provided by the ORB. A GDA object contains an object of the Global Query Manager (GQM) class, which is able to process queries, and an object of the Global Transaction Manager (GTM) class, which is able to execute global transactions. The former decomposes the query into subqueries using information obtained from the Schema Information Manager and sends these subqueries to the latter, which then cooperates with LDAs to execute the subqueries and obtain results. As soon as partial results that can be further processed are returned from the LDAs, a Query Processor Object (QPO) is created to process them. There could be many QPOs running in parallel as needed.

MIND views each data source involved as an LDA object registered with an ORB. These objects have a standard interface but, most likely, different implementations. Objects in individual databases are not registered with the ORB, that is, they are not accessible via the ORB; they are only accessible by the DBMS where they reside. For example, consider a data source storing Person information.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

With the MIND approach, the interface of Person objects is not known to the ORB. This means that MIND cannot pass Person objects around; it can only pass data that the ORB knows how to handle, such as string, integer, and so on. In general, MIND does not allow registration of fine-granularity objects. Hence, the way MIND uses CORBA is as an advanced communication backplane and is orthogonal to the technical aspects of MIND, such as schema integration and query processing.

Schema integration in MIND is performed by DBAs using an object definition language that allows specification of interfaces of objects in the global schema, and how they relate to objects exported by various data sources. Query processing in MIND aims at maximizing execution parallelism. This aspect of MIND will be further reviewed in Section 2.6.

### 2.2.4 Modeling and Language Features in FDBs

In their 1986 survey paper, Batini et al [7] investigated twelve methodologies for designing a single conceptual schema based on a set of specific user-oriented relational schemas. These methodologies are then compared on the basis of five commonly accepted conceptual schema design activities: pre-integration, comparison of schemas, conforming of schemas, merging, and restructuring. The work in this area does not address as many types of semantic heterogeneities as are known today because it is aimed at designing a conceptual schema or external view based on a set of relatively homogeneous, user-oriented schemas.

Considerable work on modeling and language features for data integration has been done in the context of federated database systems, including classifying semantic heterogeneities and proposing resolutions; identifying and designing models and/or language features that allow expression of such resolutions; and implementation of these features. Typically, primitives found in popular data models are limited in their support of data integration [41, 46]. Object data models offer more facilities but these models still need to be extended in order to be suitable for use in an FDB system. For instance, DISCO [89] extended the ODMG object model to allow a bag of extents, rather than allowing a single extent for each object class; Garlic [12] introduced the concept of *weak identities* to the ODMG object model, rather than insisting on unique and immutable object identifiers.

Previous work in this area includes [22, 67, 18, 51, 52, 66, 34, 40, 41, 45, 42, 83, 46]. These approaches differ in the canonical data model, query language, and specific language constructs provided for expressing database integration. The canonical data models employed range from relational to object-oriented and logical.

The work reported in [22, 21] uses a specific notion of generalization to facilitate integration of multiple databases. Given two types that originate from different data sources, a *generalization type* can be defined over them. The derivation of instances of this generalized type must be defined. Such derivation involves two steps. The first step is the outerjoin of the subtypes on a merge condition. This step specifies the population of the generalized type. The second step specifies how the functions on the generalized type are derived from the subtypes' function. The following example is taken



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

from [21]. Assume there are two databases, both modeling ships, with ship being modeled by type $SHIP1$ at site1 and by $SHIP2$ at site 2. A generalized type $SHIP$ is defined by the following:

```
derive SHIP from
    for s in outerjoin of (s1 in SHIP1, s2 in SHIP2)
        on merge condition ID1(s1) = ID2(s2)
      case s isin SHIP1 and notin Ship2
        ID := ID1(s)
        weight := weight1(s)
      case s isin SHIP2 and notin SHIP1
        ID := ID2(s)
        weight := weight2(s)
      case s isin SHIP1 and isin SHIP2
        ID := chooseAny(ID1(s), ID2(s))
        weight := AVG (weight1(s), weight2(s))
    endfor
end
```

The *outerjoin* of $SHIP1$ and $SHIP2$ has the following attributes: $ID1$, $ID2$, $weight1$, $weight2$. After the aggregation functions specified above are applied, $SHIP$ has two attributes: $ID$ and $weight$. A query decomposition method and various query processing tactics related to this generalization mechanism are described in [22, 21].

The Superview system [67] develops a set of *schema restructuring operators* that facilitate creation of superviews integrating the schemas of multiple databases. The operators *meet*, *join*, *fold*, *combine*, *connect* manipulate generalization hierarchy. The operators *aggregate* and *telescope* manipulate attribute hierarchy. The operators *add* and *delete* extend and reduce schema structure. This approach assumes that all participating databases, as well as the superview, are represented by a functional data model. Query processing is based on function translations.

The work reported in [18] uses *extended abstract data types* to represent data domains. These ADTs "know" how to convert a value in a native format into the canonical format. The ADTs solve the domain-incompatibility problem that is common in multidatabase queries. When values from different sources are to be compared, the ADTs are used to convert the values into canonical representation and perform the comparison. An ADT also has other functions, such as suggesting names of attributes and tables that might involve the data domain modeled by the ADT. These ADTs hold a large amount of semantic knowledge and they form a *domain knowledge base*. Another construct for representing semantic knowledge is *connector*, that describes how relations are semantically related to one another. Connectors are defined using ADTs. A relational algebra extended with connectors is then developed. This algebra includes operators such as *Delete/Add Connector*, *Delete/Add Relation/Attribute*, *Join Combine*, *Rename Relation/Attribute*, *Union Combine*, and *Difference Combine*. These operations are essentially relational manipulations enriched by the semantics of connectors. The extended algebra can be used for both schema integration and for direct manipulation of individual databases in order to formulate a query. The system takes the input



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

from the user and generates queries that access participating databases.

The two approaches described above allow creation and querying of views that hide heterogeneities. They map queries posed against such views to source queries, but do not discuss the optimization of this mapping. For instance, the impact of using ADTs on query processing is not discussed in [18]. This issue will be further discussed in Section 2.6.

The ViewSystem [40] is an object-oriented approach for querying and integrating multiple data sources, including file systems. This system allows the definition of *external classes* that physically reside in external sources. It offers several constructs for creating derived classes over existing. external classes as well as local classes. The expressive and computational power of an object-oriented data model, VODAK [26], is used in integration and query processing. Query processing in ViewSystem is a hybrid of materialization and query decomposition. Depending on the integration construct used in its derivation, a derived class defined for integration purpose decides whether and how to perform data materialization or query decomposition when processing queries posed on it. For instance, if the derivation involves aggregation functions, the class will decide to materialize, rather than decompose. If the class is derived using disjoint union, then queries will be decomposed. In the case of query decomposition, optimization issues are not discussed.

In [42], an extensive list of schematic conflicts and their resolution are given in the context of the multidatabase system UniSQL/M. UniSQL/M extends SQL with language constructs that allow creation of virtual classes which hide various types of heterogeneities.

[45] discusses language features for database interoperability; in particular, it discusses in detail the *cross-over* schema mismatches, where concepts represented as relations or attributes in one database are represented as data values in another. The paper proposes higher-order language features to express resolutions to such mismatches. The work done in this paper has significant impact on later work, such as SchemaSQL [46] that builds on the result of [45], but focuses more on making the language features compatible with SQL and establishing practical implementation techniques. [41] uses behaviors to resolve *domain* and cross-over schema mismatches. Domain mismatches arise when a concept - for instance, money - is represented differently in different sources. [41] describes language constructs in the context of Pegasus [5] that use an object-oriented database programming language to express mappings between these different representations in an integrated manner.

SchemaSQL[46] extends SQL for querying multiple heterogeneous relational databases. In traditional SQL, variables can only range over tuples in relations. SchemaSQL allows variables to range over databases in a federation, names of relations in a database, names of attributes in a relation, values in a column in a relation, as well as tuples in a relation (as in the usual SQL). This essentially makes it possible to query meta data, as well as data, a relational database. Used as a view definition language, SchemaSQL allows sophisticated restructuring of relational databases to eliminate



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

heterogeneities such as the cross-over schema mismatches discussed by [45]. A methodology for implementing SchemaSQL as a non-intrusive middleware over existing relational databases is also sketched.

A *flexible relational model* is developed in [83] for integrating data from multiple, possibly inconsistent databases. This model is based on the concept of a *cluple.* Intuitively, a cluple is a set of tuples that represent possibly inconsistent data on the same entity. That is, all tuples match on identifier attributes, but may not match on other attributes. Cluples also record other information such as the origin of component tuples, consistency status, and so on. A *flexible relation* is a set of cluples. An algebra is described to query over flexible relations. That is, selection, project, and join operations are defined over flexible relations. Intuitively, this algebra facilitates querying data that is possibly inconsistent. This is in contrast to traditional query processing approaches where inconsistencies are either ignored or assumed to be resolved completely by the use of aggregation functions. However, implementation of the flexible relational algebra is not given.

### 2.2.5 Schema and Query Translation Techniques in FDBs

In FDBs, source schemas must be translated into a *component schema* represented in the chosen canonical data model. Queries against the component schema must be translated to those against the source schema. Hence the dual issue of schema and query translation have received much attention. In FDB systems, such tasks are often performed by a translation module as part of the FDB system [5]. More recently, such tasks are performed by a system component called the "wrapper". However, recent work on wrappers mostly focuses on wrapping semi-structured sources [75, 80]; wrapping traditional sources is considered to be a mature technology. Nevertheless, this area of work is a main component in the federated database technology. This category of work includes [4, 19, 13, 39, 57, 92, 59, 93, 60, 101, 37]. A few approaches [19, 13, 57, 93, 37] concentrate on semantic enrichment, that is, to discover semantics from a given schema and represent this semantics as well as the schema itself in a semantically rich data model. Kalinichenko [39] gives a formal notion of equivalence among various data models. [60, 101] provide schema and query translations between relational and object-oriented databases. As part of the Pegasus project, [4] describes a simple tuple-an-object schema and query translation scheme. Meng et al. [59] described a general approach to schema and query translation between relational data model and data models that contain hierarchical structures, such as IMS and some object-oriented database systems.

### 2.2.6 AURORA and FDBs

All FDBs perform procedural integration, requiring a monolithic integration specification to be built manually. Constructs are provided for use in expressing resolutions and integration in these specifications, but usually no other assistance is provided. This means that FDBs typically have



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

two drawbacks. First, construction of the integration specification requires working with semantics of multiple source schemas to piece them together to form an integrated view. Without assistance, working with semantics would be difficult. Second, construction of the integration specification does not scale well; adding or removing a data source requires modification to the specification and/or to the integrated view itself. When the number of sources involved is large, such modifications will become difficult to manage. AURORA overcomes both of these drawbacks. AURORA does not require a monolithic integration specification to be constructed. A service view is pre-defined by the application requirements but the way in which data from multiple sources are combined together to support this service view is not specified by a human; instead, this integration is performed by the AURORA framework. To allow scalable integration, AURORA divides the integration process into two parts - homogenization and integration - and uses two separate mediators to perform these tasks. Homogenization requires a specification to be constructed manually by a mediator author, but it concerns only one data source. The Mediator Authors Toolkits (MATs) are provided by AURORA to assist in this construction. No mediator author needs to examine multiple data sources to piece them together, as in the FDB paradigm. Integration is performed automatically, requiring no specification to be constructed.

Most FDBs support a single canonical data model, requiring all applications using the FDB to adopt this data model. AURORA provides the applications with the flexibility of choosing a data model that best satisfies the applications' data access requirements. Currently, both relational and object data models are supported.

Technically, AURORA differs from FDB systems in that it defines new data manipulation operators and query models that are specially designed for dealing with multi-source, heterogeneous data - while most FDBs apply traditional data manipulation operators and query models to such data. AURORA's Mediation Enabling Algebras (MEAs) and the conflict tolerant query models are new techniques that have not been explored by previous FDBs.

## 2.3 Distributed Object Management

In recent years, distributed object computing (DOC) platforms such as CORBA [69] and COM/DCOM [61] have been used for managing distributed and heterogeneous applications. These platforms by themselves do not resolve the fundamental issues encountered in building IDAMSs - such as identification and resolution of structural and semantic mismatches, and efficient processing of queries against multiple, heterogeneous data sources. However, these platforms facilitate a significant level of interoperation. For instance, using an Object Request Broker (ORB), objects residing in distributed and heterogeneous environments can communicate and cooperate with one another to perform tasks that are otherwise difficult to achieve. In the context of an IDAMS, two types of objects can be



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

considered to be distributed objects:

1. System components that perform certain integration tasks, such as schema/query translation. By considering these components as distributed objects, an IDAMS can be built as a network of cooperative, distributed, and possibly heterogeneous components. Such componentization would allow the system to deal with the distributed nature of large-scale data integration gracefully. For instance, multiple data sources are often situated on machines connected by a computer network. It is desirable that wrappers and other system components reside on sites where they can function most efficiently and cooperate with one another across the network.

2. Source data objects or integrated data objects. These objects can be considered as distributed objects accessible through a DOC platform. By this means, data objects residing at various sources can be composed to form objects that are able to perform more comprehensive tasks.

An IDAMS built with a distributed object management approach supports distributed objects that are either system components or data objects. However, current IDAMSs employing a distributed object management approach mostly support system components as distributed objects. For instance, MIND [24] has an architecture consisting of system components that are built as CORBA servers. Data objects residing in various sources are not accessible beyond their home system boundary; they are manipulated by the local query processors, producing query results that are returned to the client as tuples of data, rather than objects.

In AURORA, both types of objects are considered as distributed objects. AURORA's architecture consists of components of various types; these components are built as COM components that can be identified, activated, and manipulated through COM/DCOM. As described later, AURORA's object-oriented homogenization mediator is able to return objects that are accessible through a DOC platform, and the object-oriented integration mediator manufactures integrated objects that perform methods by dispatching them to source objects that are able to perform them. Queries posed to an AURORA-OI mediator may return objects as query results; these objects can be further manipulated by the applications, in the same way objects in an OODBMS, such as ObjectStore, are manipulated. In this way, the full power of an object query language can be supported by the integration mediator.

A difficulty to be investigated in implementing data objects as distributed objects is in exporting large numbers of objects onto a DOC platform at run-time. For example, if an ORB is used as the DOC platform, a large number of objects may need to be registered and unregistered at run-time. It is not clear whether the current CORBA technology supports such activities. AURORA's work in this direction is at the framework level, as described in Chapter 6. Implementation issues are not yet studied. From a general viewpoint, a carefully designed query decomposition and optimization framework would reduce the number of objects to be registered/unregistered. However,



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

work in mediator query processing in the object-oriented mediators is also beyond the scope of this dissertation.

## 2.4 The Mediator Systems

Recently, many systems have been built with the *mediator architecture* [91]. A mediator is a middleware system that satisfies certain language and interface requirements, so that it can be composed with other mediators to perform more complicated tasks. In this sense, FDBs are mediators if they are capable of accessing other FDBs and are open to being accessed by other FDBs. The difference between a mediator and a FDB is not in what they do and how they do it, but in what they are used for. Originally [91], mediators were used to provide domain knowledge; they had to be able to express such knowledge, and exchange knowledge with other mediators using a common language - often a knowledge representation language, a common ontology, or logical rules. Consequently, many mediator systems built for data integration use logic for integration and query processing. FDBs are often used to provide a *virtual database*, a database that appears to be a traditional database to the application, but really gets its data by combining data residing at relevant sources. Consequently, most FDB systems use a SQL/OQL-like language for data integration, and process queries using an algebraic framework such as relational algebra and object algebras. Both mediators and FDBs can be considered as IDAMSs. It is conceivable that a FDB be built as a mediator system. It is also conceivable that an FDB act as a mediator in a mediator architecture. There is no fundamental reason for a mediator and a FDB to employ different underlying technologies. It is for historical reasons that mediators often employ a logic-based framework for integration and query processing, while a FDB's underlying framework is similar to that of a traditional DBMS.

### 2.4.1 TSIMMIS

The TSIMMIS project at Stanford [74, 75, 73, 72] represents a large step away from most previous work. Rather than a semantically rich, structured data model, TSIMMIS uses a self-describing model - the *Object Exchange Model*, OEM - for expressing integration and for querying. OEM is an information exchange model; it does not specify how objects are structured, it only specifies how they are sent and received.

In TSIMMIS, one does not need to define in advance the structure of a source object of interest, and there is no notion of schema or object class. Each object instance contains its own schema, it is *self-describing*. An OEM object consists of four fields: an *object id*, a *label* which explains its meaning, a *type*, and a *value*. Fields that are not important are omitted from the representation(as is often the case in this section). The following OEM object describes a person object. This object has three components, $component_{1-3}$, representing the name, office number, and the department of



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the person described. The object given below has name "Fred", office number 333, and he works for the *Toy* department:

< p1, *person-record*, *set*, {$component_1$, $component_2$, $component_3$, } >
< $component_1$, *name*, *string*, *"Fred"*>
< $component_2$, *office-number-in-building-5*, *integer*, *333*>
< $component_3$, *department*, *string*, *"Toy"*>

Each data source to be accessed is viewed as a collection of OEM objects in the above form, with no predefined structure. Querying in OEM is via *patterns* of the form *<object-id, label, type, value>*, where constants or variables can be put in each position. When a pattern contains constants in the label (value) field, it matches successfully only with OEM objects that have the same constant in their label (value). For instance, the following pattern would match successfully with person Fred given earlier:

*<person-record, {<name "Fred">, <department "Toy">}>*

Essentially, this pattern matches with all *person-records* that have a component *name* with value "Fred" and a component *department* with value "Toy". Notice that this pattern matching assumes no structure on the objects, as long as the object has the right label with the right value, it matches successfully. This effectively makes the labels (*person-record*, *name*, *office-number-in-building-5*, *department*) first-class citizens. Labels do not put any constraints on what types of queries are acceptable, rather, they can be queried themselves.

Queries and view specifications in TSIMMIS are also formed using patterns. The TSIMMIS Mediator Specification Language (MSL) is a rule-based language. For instance, the following rule defines a view *ToyPeople* that contains names of all people who work in the Toy department:

*<ToyPeople, {<Name N>}>:- <person-record, {<name N>, <department "Toy">}>*

The following query finds all persons who have name "Fred":

*FredPerson :- FredPerson:<person-record, {<name "Fred">}>*

In this query, *FredPerson* is an *object variable*. The formula to the right of :- says that *FredPerson* must bind to all *person-records* with a sub-object by the label of *name* and value of "Fred". The symbol :- says that all such objects are included in the query result. Notice that the query result is potentially heterogeneous, with objects having all sorts of structures, except that each object must have a label *person-record* and a *name* sub-object with value "Fred".

All data sources in the access scope must be covered with a TSIMMIS wrapper. TSIMMIS provides a *wrapper implementation toolkit* to support fast generation of wrappers. These wrappers are indeed an OEM query processor. The wrapper implementer is required to (1) describe the



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

types of OEM queries that the source can handle using query templates; and (2) map these query templates to local queries/actions at the data source.

Intuitively, OEM is flexible enough to represent data of any type, from unstructured random records, to relational data, to complex objects. After all types of data are represented in OEM, they can then be integrated. The TSIMMIS approach uses logic rules that transform and merge OEM objects from various data sources to form a mediator view. This view can then be queried. Query processing in TSIMMIS leverages deductive database techniques; it includes view expansion and execution plan generation. In [74], various aspects of the OEM model are defined and discussed. In [75], an approach for developing OEM wrappers for semi- or unstructured data sources is described. In [73], an OEM-based mediation language and its implementation is described. This language allows creation of integrated views in the mediator that removes various types of semantic conflicts. In [72], an approach for object matching (referred to as object fusion in this paper) using OEM is described. This approach allows resolution of instance level conflicts. An approach for global optimization of queries posed against these "fused" objects is also described.

In the database community, OEM is also the representative of an emerging data model that is not constrained by database schemas. This feature alone removes a major representational heterogeneity among data sources. The labeled-tree structures like those in OEM can represent all sorts of data structures equally well and have great potential in supporting integration of heterogeneous data. Query and manipulation language, and optimization techniques, are being developed for this new data model [11].

### 2.4.2 HERMES

HERMES [87] is a mediator system that uses a logical model for integration and query processing. In a HERMES mediator, a data source to be accessed, called a *domain*, is modeled as a triple $< \sigma, \mathcal{F}, \mathcal{R} >$, where $\sigma$ is a set of values, $\mathcal{F}$, a set of functions that the domain is able to perform. and $\mathcal{R}$, a set of relations over elements of $\sigma$. For example, for a relational data source, $\sigma$ consists of all the tables as well as individual values stored in these tables; $\mathcal{F}$ includes the usual relational operators project, select, and join, and $\mathcal{R}$ is a set of predicates over the tables. *Domain calls* to a domain retrieves data from this domain. For example, a *domain call* to the relational domain *PARADOX* could look like this:

$$PARADOX : project('parts', \text{``}partid\text{''})$$

This domain call asks the *PARXDOX* relational database to perform a project on table "parts" on attribute "partid". A *domain call atom* is formed by a small set of predicates taking domain calls as input, for example:

$$is(\{\text{``}green\text{''}\}, PARADOX : project('parts', color))$$



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

evaluates to true if all objects in *parts* have color green. A *mediatory rule* is of the form:

$$A_0 : [\mu_0, \tau_0] \leftarrow A_1 \& ... \& A_n$$

where $A_i$'s are domain call atoms and $[\mu_0, \tau_0]$ is an annotation on uncertainty and time. In HERMES. these rules are used to perform query, information extraction, information merging (called pooling). and conflict detection/resolution over all the imported domains. A domain call caching method is developed in [3] to improve query performance in an environment that involves distributed and autonomous data sources.

HERMES supports domains of various types, including relational, spatial, text, and pictorial. In order to set up a domain for a new data source, that is, to *import* a data source into HERMES, a *mediator author* specifies a set of domain functions that can be accessed by the mediator, designs a data structure to be used to hold the output of these functions, and implements procedures for parsing the output properly to fill this data structure. Once imported, the HERMES mediator language is used for extracting and merging information from multiple domains. This language is based on mediatory rules (as described above) and has a Prolog-like syntax.

HERMES provides a mediator programming environment (MPE) that assists mediator authors in constructing mediators. The task of constructing a mediator includes *domain integration*, that is, importing of data sources, *conflict resolution* and *information pooling.* MPE provides toolkits for all three tasks. The conflict resolution toolkit is interesting, and works as follows. The mediator author specifies integrity constraints that disallow inconsistent data values. The toolkit, upon receiving this constraint, generates all possible violations and, for each violation, it asks the mediator author to provide a resolution. A set of commonly used resolution strategies is also provided.

### 2.4.3 AURORA and Mediator Systems

The original mediator architecture prescribes mediators providing data/knowledge services of specific kinds. These specialized mediators can then be composed to provide more comprehensive services. Most mediator systems developed for data integration, such as TSIMMIS and HERMES, provide a single type of mediator that is used to support services of different kinds. Such mediators must provide a framework generic enough to achieve all kinds of data integration tasks. In contrast, AURORA mediators are specialized; they are designed to perform either homogenization or integration. The frameworks employed by individual mediators are small, specialized, and easy to use. This approach also allows enabling techniques tailored for specific integration tasks to be developed.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

## 2.5 Paradigms For Dealing with Highly Dynamic Environments

With the advent of the Internet and WWW, more and more information repositories have been opened. These data sources are highly dynamic with varying query processing capabilities. The usual approach of "understanding" each data source, integrate them, and then process queries is not viable in this context due to the large number of sources present. For instance, federated database systems often provide insufficient support for integrating a large number of data sources that varies in availability. Earlier data integration systems also did not deal with semi-structured or unstructured data sources. A few systems have been built to work with highly dynamic environments. These systems typically provide features specially designed to cope with one or more of the problems present in a dynamic integration scope.

DISCO [89] works with a set of dynamic data sources by facilitating easy hook-in of individual sources, and by dealing with unavailable data sources at query processing time. It extends the ODMG IDL to allow a bag of extents for a single interface type. Thus, adding a new source is done by adding a new extent into the bag of extents of a global class. DISCO also proposes an approach for dealing with unavailable data sources.

The DIOM system [53] accommodates the dynamic changes of the environment by identifying relevant data sources, determining how these sources work together to provide data of interest, and binding to each of them at query time. This is a query mediation approach similar to [77] but with different underlying techniques. DIOM's query processing engine uses an algebraic framework to manipulate source data, while [77] uses logic.

The Information Manifold project [50, 48, 49] considers large number of data sources with varying query capabilities. Assume there is a *worldview* at the mediator level. Most likely, this is a virtual view; it is the way that the target applications would like to see the world. Each data source to be accessed can be regarded as a materialized view of this worldview, but with *capability records* attached describing the types of queries it can handle. Thus, the problem of answering a query against the worldview is transformed to that of answering a query with existing materialized views, with additional constraints. This problem is solved in [50] in relational context. Recently, this work has been extended by [62], which allows expressing the materialized views using SchemaSQL. This increases the ability of IM in dealing with structurally heterogeneous sources. Other related work is the query folding approach [76], which allows queries to be answered using existing resources such as materialized views, cached query results, or queries answerable by an existing query processor. Integration of multiple sources in these approaches is scalable; it only means addition of a new materialized view. Handling sources with limited query capabilities is a very useful feature in accessing a wide range of information repositories such as those typically present on the Web.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

All of the above-reviewed systems have no or limited capabilities in dealing with instance level conflicts. DISCO and IM assume that data from various sources are consistent. For DISCO. dealing with instance level conflicts might be a matter of introducing more model/language extensions to the ODMG model, but that is yet to be seen. It is not clear whether the underlying framework of IM is able to deal with such conflicts. It is also not clear whether or how DIOM deals with such conflicts.

### 2.5.1 AURORA in Highly Dynamic Environments

There are two equally important dimensions to the problem of integrated data access in a highly dynamic environment: the heterogeneities among the sources, and the sheer number of the sources. It is difficult to support both dimensions. Systems with more traditional paradigms such as FDBs and some mediator systems provide elaborate facilities for dealing with heterogeneities, but their support for scalability is often insufficient. Newer systems geared towards integrating a large number of dynamic sources are often weak in the facilities for dealing with heterogeneities. For instance, a rich framework for value and structural conversion of source data is missing from IM, DISCO, and DIOM, and none of these systems provides tools assisting in working with semantics. Moreover, none of these systems deals with instance level conflicts.

Similar to IM, DIOM, and DISCO, AURORA is built with the goal of supporting large-scale data integration in a highly dynamic environment. However, AURORA does not suffer from the problems described above, as these systems do. With the 2-tiered mediation model, AURORA is able to support scalable mediation without neglecting dealing with heterogeneities. This is achieved by dividing the data integration into two sub-tasks: homogenization and integration. Homogenization mediators deal with a wide range of semantic and structural heterogeneities. Conflict tolerant query models supported by the integration mediators allow instance level conflicts to be dealt with gracefully.

## 2.6 Query Processing and Optimization in IDAMSs

Algebraic query optimization is an important form of optimization and is the basis for cost-based query optimization techniques. In the context of query processing for IDAMSs, algebras that are suitable both for manipulation of heterogeneous data, and for use by a query optimizer, are of special interest. Query optimization techniques for IDAMSs are also relevant. In this section, previous work is reviewed with this perspective in mind.

Like traditional DBMSs, an IDAMS that supports queries over multiple sources, either via a global schema or via a multidatabase query language, relies on an algebra that transforms and integrates data from multiple sources. There are many levels at which this algebra can be discussed.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Of interest to this thesis is the **enabling algebra**, the lowest level algebraic framework supporting query processing: it must be simple, so that it can be manipulated by a query optimizer, but it must also be expressive, so that it can be used to represent all the logical operations required for transforming and assembling data. Operators in this algebra are referred to as the **enabling operators**. For example, the enabling algebra in a centralized relational database usually consists of selection, project, join, and union operators. In distributed relational databases, the algebra is extended to include the semijoin operator, in order to manipulate distributed data efficiently. [9] proposed an operator, "materialize", for manipulating data in an OODB; this operator can be transformed into a join and is believed to open up more query evaluation alternatives.

When processing a query that involves multiple data sources, the query is first translated into an expression in the enabling algebra that only references imported objects. This expression is then optimized and evaluated. In general, query processing and optimization in an IDAMS consists of the following categories of issues:

1. Identifying the enabling algebra. Other than the usual data manipulations known to traditional DBMSs, data integration may require new data manipulation operators to be developed. For instance, one such operator has been identified and studied as the *semi-outer-join* operator, proposed by [20], to allow efficient processing of queries posed against generalized types. This operator will be further discussed later.

2. Algebraic rules for transforming expressions in the enabling algebra. Even if one assumes that the enabling algebra of an IDAMS is no different from that of distributed database systems, algebraic transformation raises new issues. For instance, in multidatabase query processing, outerjoins followed by aggregation functions are required.(This combination is referred to as generalization [21]). These are not new operators but they are expensive to evaluate in the context of IDAMSs. In general, new transformation rules must be added to accommodate any new enabling operators developed, and to transform expensive combinations of existing operators (e.g., outerjoin and aggregation) into less expensive ones.

3. Cost modeling of global execution plans. Computing the cost of a query execution plan requires knowledge about data volume, distribution, indices available, processing speed of various external sources, and the speed of communication links. Early work assumes these parameters are available in the context of IDAMSs [10, 99, 38]. In reality, autonomy of data sources determines that the parameters required for cost modeling are not necessarily available. and the data sources are not as cooperative as sites in a distributed database. For instance, semijoins between autonomous DBMSs may not be as efficient as in distributed databases, since it may be impossible to send data directly into the system buffers of the query processor of an autonomous DBMS [56]. More recently, research in this regard attempts to cope with un-



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

available knowledge for cost estimation by calibrating, sampling, and dynamically adjusting existing cost estimations. Garlic [12, 35, 79] also builds facilities that allow wrappers to export cost information. These techniques indeed cope with specific situations encountered by IDAMSs.

In the rest of this section, previous work is reviewed along the three dimensions described above. The first two dimensions are relevant to the work on AURORA reported in this thesis. The third is relevant to AURORA in general, but is not relevant to the work reported here. A brief review of work in this dimension is included for completeness.

### 2.6.1 New Enabling Operators and Evaluation Strategies

[20] describes the use of *semiouterjoins* to process queries over generalized hierarchies in multidatabases. The focus is on *generalized types* that involve aggregation functions. Consider the ship example, as given in Section 2.2.4, and a query "*select all SHIP that have weight of at least 55*". To process this query correctly, one could first compute the *weight* attribute of each ship. This requires fetching complete populations of $SHIP1$ and $SHIP2$ from the respective sites they reside into the global site; this operation can be very expensive. The *semiouterjoin* of $SHIP1$ by $SHIP2$ partitions $SHIP1$ into two parts: the *private part* and the *overlap part*. The overlap part contains ships that are also described in $SHIP2$. This part must be sent to the global site to be further evaluated. One can apply local selection $weight1 > 55$ at site1 on the private portion of $SHIP1$. Only the $SHIP1$ records that satisfy this condition are sent to the global site. After the semiouterjoin is completed at site1, the private or overlap part of $SHIP2$ can be sent back to site2 and similar procedures take place there. Sometimes, local selection can even be performed on the overlap part as well, but this depends on the type of aggregation functions used. This range of techniques are also described in [20] and surveyed in Section 2.6.2. Hwang and Dayal [38] developed general algorithms for identifying optimal schemes of using semiouterjoin for improving the performance of projection, selection, and join over generalized types involving an arbitrary number of types. These algorithms assume that all cost measures useful for cost modeling are available.

Goldhirsh and Yedwab [32] suggest that the traditional query modification approach is inappropriate for optimizing queries that involve *generalized types* (Section 2.2.4). Consider a query involving type Person that is the generalization of types Student and Employee. The traditional approach would always modify the query so that it can be decomposed into subqueries against types Student and Employee. This paper argues that in a distributed environment, such queries can sometimes be more efficiently processed by materializing Person rather than by performing query modification to eliminate it. It is not necessary to materialize all the attributes of type Person. Inclusion of the materialization-based query evaluation plans into the optimization space is also discussed.

A few papers generated from project Mermaid (more recently known as Interviso [88]) are closely



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

related to general query optimization techniques in distributed relational database systems. These techniques always assume that statistical and system measures required for cost modeling are available. In the context of Mermaid, [10, 99] describe the general query processing strategies: semijoins and replication. The cost model is extended to include these processing strategies, which are variations of SDD-1 algorithms and replication methods for query processing in distributed database systems.

In [100], Yu et al. suggested two ways to improve query processing. First, use semantic knowledge to remove unnecessary operations or simplify certain operations. Second, compare actual runtime numbers such as relation sizes, data transfer rate, and processing cost with those estimated by the static cost formulas, and update these formulas when they deviate from the actual numbers drastically and consistently.

### 2.6.2 Algebraic Transformation Rules

As far as query processing is concerned, the most studied data manipulation operator is generalization (Section 2.2.4), which is usually equated to outerjoin followed by aggregation. An example of generalization is the $SHIP$ example discussed earlier. In that example, $SHIP$ is derived as the outerjoin of $SHIP1$ and $SHIP2$ on their IDs, followed by aggregation functions. Both outerjoin and aggregation functions are expensive to evaluate. In [20], Dayal described various algebraic tactics to transform such expressions into less expensive ones. The main objectives of such transformations are to distribute selections and joins over generalization, and to use semiouterjoin reductions discussed in Section 2.6.1. Distribution of joins is based on the rules that distribute selections. The distribution of selections is discussed in more detail below.

Whether a selection can be distributed over a generalization is determined by whether the following equation holds:

$$\sigma_{A\theta a} AGG_A(R_1 \overline{\infty} R_2) = \sigma_{A\theta a} AGG_A(\sigma_{A_1\theta_1 a}(R_1) \overline{\infty}\, \sigma_{A_2\theta_2 a}(R_2))$$

where $\overline{\infty}$ stands for outerjoin, $AGG$ is an aggregation function, such as *sum*, *average*, *min*, and *max*. $AGG_A$ means that this aggregation function is used to derive value for attribute $A$. $A_1$ and $A_2$ are attributes in $R_1$ and $R_2$, respectively. The meaning of expression $AGG_A(R_1 \overline{\infty} R_2)$ is the following: for each tuple $t \in (R_1 \overline{\infty} R_2)$, derive a new attribute $A$ whose value is computed as $t[A] = AGG(t[A_1], t[A_2])$. The attribute *weight* in the $SHIP$ example is derived this way. In [20], the following rules are given to push the selection across $AGG$ and $\overline{\infty}$:

- If $AGG = chooseany(A_1, A_2)$, true distributivity holds; that is, the above equation holds with $\theta_1 = \theta_2 = \theta$.

- If $AGG = max(A_1, A_2)$, there are three cases:



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. If $\theta$ is $>$, then true distributivity holds; that is, the above equation holds with $\theta_1 = \theta_2 = \theta$.
2. If $\theta$ is $=$, distributivity holds with both $\theta_1$ and $\theta_2$ being $\geq$.
3. If $\theta$ is $<$, no distributivity at all.

- If $AGG = min(A_1, A_2)$, there are three cases:

1. If $\theta$ is $<$, then true distributivity holds; that is, the above equation holds with $\theta_1 = \theta_2 = \theta$.
2. If $\theta$ is $=$, distributivity holds with both $\theta_1$ and $\theta_2$ being $\leq$.
3. If $\theta$ is $>$, no distributivity at all.

- If $AGG = count(A_1, A_2)$, or $AGG = sum(A_1, A_2)$, or $AGG = average(A_1, A_2)$, no distributivity at all.

A more complete and extensive set of rules are given in [58]. The main extensions are in developing distributivity under certain conditions when $AGG$ is *sum* or *avg*. For instance, for the case of $AGG = sum(A_1, A_2)$ and $\theta$ is $=$, the above equation holds with both $\theta_1$ and $\theta_2$ being $\leq$ if it is known that $A_1$ and $A_2$ values are not negative. These rules help in further reducing the amount of data that must be transferred to the global site where aggregation finally takes place. However, the applicability of these rules is generally limited by the type of the aggregation function and the selection predicate.

Chen [15] describes techniques to optimize outerjoins when no aggregation function is present, that is, when data inconsistency is not present. Essentially, this work considers the simpler cases of generalization, without aggregation functions. Let $R = R_1 \overline{\bowtie} R_2$. $R$ consists of three partitions: (1) the join of $R_1$ and $R_2$; (2) $R_1$ tuples that have no matching tuple in $R_2$ padded with null values; and (3) $R_2$ tuples that have no matching tuples in $R_1$ padded with null values. A *one-sided outerjoin* is the union of (1) and (2) or (1) and (3). If both (2) and (3) are empty, the outerjoin becomes a regular join. Regular joins are cheaper to compute than one-sided outerjoins, which are in turn cheaper than outerjoins. In [15], Chen described rules that use the cheapest strategy to process a query involving $R$. For instance, a global query referencing $R$ can be processed without involving $R_1$ at all if (1) no attribute of $R_1$ is involved in any predicates; and (2) all target attributes can be found in $R_2$.

### 2.6.3 Coping with Unavailable Optimization Information For Cost Modeling

When accessing multiple autonomous database sources, information about source data may not be available. Such information may include data sizes, selectivity, distribution, and any available fast access paths such as indices. Different query processing and optimization tactics may be used in



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

different sources. These factors make it difficult to identify an optimal execution plan that involves multiple sources. Autonomy also makes the cooperation of the global query processor and the local query processor difficult. A few approaches exist to cope with this situation. Lu et al. [56] proposed to monitor subquery execution and compare the cost estimate of the global query optimizer with the actual cost. The result of this comparison is then used for improving the global cost model and for adjusting query execution plan at run-time. Du et al. [25] proposed a calibration method to deduce the cost formulas for a given database. Assume that the cost of queries can be modeled by a set of formulas with unknown coefficients. In order to derive these coefficients, a specially constructed calibrating database is loaded into the local database and a set of queries are run against it. Cost metrics of these queries are recorded and are used to deduce the unknowns in the cost formulas.

MIND, a multidatabase system reviewed in Section 2.2.3 develops some techniques in global query optimization [27, 71]. These include:

1. Cost-based global query optimization in case of data replication. This technique deals with site selection issues in cases when a subquery can be executed at more than one site.

2. Cost-based inter-site join optimization. This technique starts from a left-deep join tree and attempts to transform this tree into a more bushy tree so that response time can be reduced by exploiting parallelism. However, it is stated in [27] that the performance study performed shows that the gain in performance cannot compensate for the complexity of rearranging the tree to maximize parallelism.

3. Dynamic optimization of inter-site joins. This technique is still cost-based but is dynamic in that it uses partial results at run time, adjusts cost estimation and determines the next step. This approach reduces uncertainties in cost estimation.

Garlic [12] has produced significant work in cost modeling when querying diverse data sources [35, 79]. This work extends the rule-based query optimization technique proposed by [55], by providing a framework for wrappers to export cost information to a degree chosen. In particular, wrappers export such information using *Strategy Alternative Rules*, which are fired to produce alternative plans and their costs. These plans are then evaluated by the Garlic query optimizer to choose the optimal plan. The main appeal of this approach is that it provides wrappers with guidelines for exporting cost information, and it also allows them to evolve to provide more information as it becomes available. The Garlic query optimizer then takes the information provided by wrappers into consideration at query processing time. This approach, combined with the calibration techniques evaluated earlier, provides a good basis for query processing in IDAMSs.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

## 2.7 Summary

In this chapter, a survey of previous work is given, so as to position AURORA in respect to them. AURORA differs from previous work in paradigm, mediation models, and the type of techniques developed. Most of these changes in approaches are made to improve usability, scalability, and efficiency. In the next chapter, the overall architecture of AURORA is given. A road-map to the techniques developed in AURORA will also be provided.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# Chapter 3

# The AURORA Architecture

This chapter describes the AURORA architecture. To start with, a detailed discussion on the design principles of the two-tiered mediation model is given. The AURORA architecture is then described, including an overview of AURORA mediators and how they work together. Finally, a road-map into the enabling techniques is provided; these techniques will be the subjects of the later chapters.

AURORA mediators perform a specific type of mediation - that of integrating data from multiple heterogeneous sources. This is the only type of mediation of interest to AURORA. To distinguish this type of mediation from the general mediation concept, the term *data mediation* is used. A *data mediation system* is a system that performs data mediation. By nature, a data mediation system is a specific kind of IDAMS - the kind built with mediator architecture. In the rest of this dissertation, the term *data mediation system* is used to refer to the middleware systems to be constructed using AURORA mediators.

## 3.1 AURORA Mediation Model

The ultimate goal of data mediation is to support a service view determined by the target applications' data access requirements. To the applications, the service view is a global database schema that is designed to suit their data access needs. AURORA mediators are responsible for providing data according to this view, by transforming and combining data from participating sources. Fundamentally, AURORA has to achieve two things: to remove heterogeneities among participating sources, and to combine source data in a meaningful way. These tasks are accomplished differently by different data mediation systems.

A *mediation model* defines the tasks to be completed in a mediation effort, and the relationships among these tasks. AURORA's mediation model is based on a perception of the heterogeneities encountered when integrating heterogeneous data sources and how they should be handled; and a perception of the relationship between source schemas and a target service view. The AURORA



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

mediation model, shown in Figure 1.1, is a two-tiered model. It models data mediation as a two-step process: *homogenization* followed by *integration*, performed by specialized mediators.

### 3.1.1 Two-tiered Mediation Model

The two-tiered mediation model is based on the following perceptions of the heterogeneities encountered during integration of large number of data sources and on the relationship of source schemas to the service view:

**Heterogeneities.** Two categories of heterogeneities must be dealt with when integrating multiple data sources: *schematic mismatches* that arise when the same application domain is modeled differently by different sources; and *instance level conflicts* that arise when inconsistent values on the same application entity are recorded by different sources. Intuitively, the first type of mismatch happens because sources "use different languages to talk about the same thing". The second type happens because sources "say different things about the same matter".

**Relationship between participating sources and the service view.** A data source participates in a service view when it is able to contribute data on *some* aspects of *some* objects of the service view. This assumption covers most, but not all, scenarios of data integration. For instance, this assumption excludes the scenarios where the age attribute of a person in the service view is derived by taking the average of all the age values of this person provided by various sources. In this case, no data source is perceived as contributing the "correct" age value on entity Person. Such use of aggregation functions is often, not always, for the purpose of resolving instance level conflicts. In AURORA, instance level conflicts are treated using a conflict tolerant query model, not aggregation functions, as described in Chapter 5.

Data integration is achieved in AURORA through the following steps:

1. **Wrapping.** Build a wrapper around each data source so it "speaks" in a data model and query language that can be understood by AURORA mediators.

2. **Homogenization.** Derive a view on top of each data source. This view conforms to the service view in both structure and semantics, and is referred to as the *homogenizing view*. This view describes *some* aspects of *some* of the objects in the service view. To derive this view, all schematic mismatches between the sources and the service view must be resolved. This process is referred to as the *homogenization* of the source. In AURORA, specialized *homogenization mediators* support homogenization.

3. **Integration.** Devise a mechanism to answer queries against the service view using data contributed by various sources through the respective homogenizing views. To do this, instance level conflicts must be resolved. This process is referred to as *integration*. This process is supported by specialized mediators, the *integration mediators*.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Wrappers are data model/language translators; each makes a data source accessible to AURORA mediators. However, wrappers do not deal with semantic or structural heterogeneities. A more detailed discussion on AURORA wrappers is given in Section 3.2.1. Technology for wrapper construction already exists and is not a focus of AURORA research; wrapper construction is treated as an engineering issue in AURORA. The two tiers of mediation refer to homogenization and integration.

### 3.1.2 Building A Data Mediation System with AURORA: A Scenario

Assume that a class of applications needs to access data residing at various data sources $S_1, ..., S_n$, through a service view $V_{service}$, expressed using data model $D_{service}$. Also assume that $V_{service}$ does not change but the list of sources is dynamic, new sources may be included, and previously included sources may decide to not allow their data to be accessed by this class of applications. Once the participating sources are identified, wrappers must be constructed for these sources. Wrapping can be done independently and in parallel for each source. Wrappers must support a relational or ODMG interface - whichever is most easily generated. Choice of a wrapper data model should be independent of the data model employed by the target service view; AURORA is responsible for accessing wrappers of various kinds. At the same time, a mediator author chooses an integration mediator, $M_V$, that supports data model $D_{service}$, and initializes it with $V_{service}$. Once initialized, $M_V$ accepts application queries immediately although its access scope may be insufficient as far as the applications are concerned, since $M_V$ does not access any data source, directly or indirectly, upon initialization. To expand the access scope to include desirable sources as fast as possible, one or more mediator authors can be assigned the task of using a homogenization mediator supporting data model $D_{service}$ to homogenize the data sources previously wrapped. After being homogenized, a source informs the integration mediator $M_V$ of its existence and is included in the access scope of $M_V$ automatically.

More scenarios using different types of AURORA mediators to construct a data mediation system are given in Section 3.2.2. The mediation model, as described above, is designed to facilitate scalable data integration, where adding and removing a source from the integration scope is easy. This is discussed further in the next section.

### 3.1.3 Two-tiered Model and Scalability

To include a new data source in the access scope of a data mediation system, two issues must be resolved:

1. *Communication.* It must be possible to "talk" to the data source. This is achieved by a wrapper that removes idiosyncrasies of the data source in communication protocol, data model, and



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

query language.

2. *Semantic integration.* It must be possible to include the data source in the access scope of an integrated view.

Scalable mediation should make adding or removing a data source to/from the access scope easy to do. This in turn requires fast *wrapper generation* and *scalable semantic integration*, which requires the following:

1. Sources can be added into, and removed from, the access scope without causing previous integration effort to be obsolete.
2. Adding and removing a data source from the integration scope should be made as simple as possible.

Various enabling techniques have already been developed for rapid wrapper construction [75, 80]. As discussed in Chapter 2, previous work does not provide satisfactory support for scalable semantic integration.

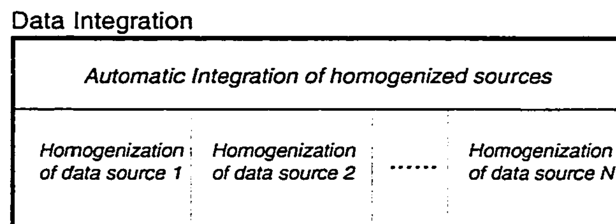



Figure 3.1: Divide-and-conquer data integration of AURORA

AURORA facilitates scalable data integration by prescribing a divide-and-conquer approach towards data integration. As shown in Figure 3.1, a data integration task is divided into $N + 1$ subtasks, where $N$ is the number of participating data sources, including $N$ homogenization tasks and 1 integration task. The $N$ homogenization tasks can be performed in parallel and independently of one another. The integration of homogenized data sources is performed automatically by AURORA. Homogenization of single data sources is significantly simpler than any integration effort that requires examining multiple sources at once. Moreover, AURORA makes homogenization more manageable, and hence potentially faster, by providing tools to assist in this process. Participating sources register the data they provide through respective homogenization mediators; this data will be integrated by the relevant integration mediator automatically. Removing a source from the access scope only requires the relevant registrations to be cancelled.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

### 3.1.4 Why Two-tiered?

The two-tiered model defines a divide-and-conquer approach to information integration. Such an approach facilitates applications, such as electronic commerce, that require access to large numbers of diverse data sources. It also allows the data mediation system to better manage the technical complexity in large-scale middleware. An electronic commerce example is given below to illustrate these points.

#### Two-tiered Mediation in Electronic Commerce

A virtual shopping mall is a typical electronic commerce (EC) application. A key component in this application is the catalog system. Companies organize their catalogs differently; this gives rise to a set of heterogeneous and autonomous catalogs. When the number of participating catalogs is large, it is difficult for a shopper to locate items of interest. One approach is to require all vendors to re-organize their catalogs into a common format and merge them into a central catalog that allows customers to perform sophisticated searching without dealing with individual catalogs. This requires re-engineering of existing catalogs. In general, vendors want to participate in the central catalog without making changes to their existing ones. A *virtual catalog* that has the look and feel of a central catalog but holds no physical data, is desirable. Upon a customer request, this catalog retrieves relevant information from (multiple) individual catalogs and assembles an answer. Such a virtual catalog should satisfy the following requirements: (1) it is up-to-date, but does not violate the autonomy of the participating catalogs; (2) its search performance does not degrade as the number of participating catalogs increases; (3) it allows easy inclusion of new catalogs and integrates with other EC applications; and (4) it is easy to construct; tools should be provided to assist in this process.

Typically, to include a supplier catalog in a virtual catalog, the supplier is first required to map his or her catalog into a format required by the virtual catalog. Essentially, the supplier catalog must be *homogenized* before participating in the virtual catalog. Homogenization is performed by suppliers independently, referencing the common catalog format. Individual suppliers are not concerned with inter-catalog conflicts, which are resolved at the central catalog level. Often, suppliers are provided with a *workbench* to perform homogenization. This workbench is a homogenization mediator, while the central catalog is an integration mediator. A supplier can participate in multiple virtual catalogs requiring varying catalog formats. In this case, the supplier must use multiple homogenization mediators.

AURORA's two-tiered mediation model closely corresponds to the process of constructing virtual catalogs. A mediation model suitable for building virtual catalogs must clearly define which mismatches are to be resolved by the suppliers independently, and which mismatches are to be handled at the central catalog level. Suppliers are responsible for removing mismatches between their



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

catalog and the virtual catalog schema, and the virtual catalog is responsible for resolving instance level conflicts, such as the same product bearing different names.

A virtual catalog effort can be initiated by a third-party broker who seeks to offer value-added catalog services using AURORA mediators. The broker first designs a common catalog structure, its data model and query language. To include a vendor in the virtual catalog, the broker homogenizes the vendor's catalog using an AURORA homogenization mediator. This process maps the vendor catalog structure and semantics into those in the common catalog. After homogenization, it should be straightforward to "plug" a catalog into an AURORA integration mediator that supports the common catalog. While homogenization is a more complex process, the broker can hire a few people to homogenize individual vendor catalogs in parallel. An integration mediator is where large number of virtual catalogs merge but the integration is a simple mechanism. Overall, construction of the virtual catalog is scalable.

### Managing Complexities

Integrated access to a large number of highly heterogeneous data sources is complicated. There are two aspects to this complexity: integration and query processing.

**Complexity in integration.** When there are 100 sources involving many types of mismatches and conflicts, which one should be resolved first? Can several people work on the same integration task? What kind of assistance is provided for working with semantic heterogeneities? Most previous work focused on classifying mismatches and proposing resolutions, without prescribing the sequence in which these mismatches are to be identified and resolved. Only systems that perform declarative integration allow several people to work on the same integration task. Assistance for working with semantics provided by these systems is insufficient.(For instance, neither IM nor DISCO provide such assistance.)

**Complexity in query processing.** When a large number of highly heterogeneous sources are involved in a query, there arises a complex optimization problem that is unknown to traditional data management systems. Query optimization in middleware systems is known to be a difficult problem even without considering the scale of the system [56]. In large scale middleware systems such as virtual catalogs in EC, this problem is even more difficult, as discussed in Section 2.6.

AURORA's two-tiered model enables better management of both complexities. AURORA's divide-and-conquer data integration approach helps in managing the complexity of integration. In query processing, AURORA's two-tiered mediation model enables the decomposition of the query processing issue into two smaller problems: query processing in homogenization mediators and in integration mediators. As shown in Chapter 4, each type of AURORA mediator uses a specialized Mediation Enabling Algebra (MEA) to facilitate efficient query processing.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

## 3.2 AURORA Architecture

This section describes the general forms of AURORA mediators, including data model, query language support, interfaces, and also how they work together to facilitate data mediation.

### 3.2.1 Data Sources and Wrappers

Data sources can be of any type but they must be covered with a wrapper that facilitates accessing of the source through an ODMG [14] interface or a relational interface, whichever is most easily generated. ODMG has the modeling and querying power to manipulate and query relational data as sets of "structs". Such OQL queries can be translated to SQL queries in a straightforward manner. Generally, AURORA's homogenization mediators, which are the clients of wrappers, can access either ODMG sources or relational sources. It should be possible to wrap up sources as read-only or updatable. Only the read-only wrappers are considered in AURORA.

AURORA employs "thin" wrappers in that the schema presented to the world by a wrapper is a normal relational or ODMG schema with no restrictions. Wrappers do not perform any semantic translation, but only syntactic mappings that make the data in the source accessible in the relational or ODMG model. Wrapper generation issues are not investigated within the AURORA project; others have investigated this problem [75, 80].

AURORA uses commercial middleware products to build wrappers. As an example of such a wrapper, consider a commercially available middleware system - the ISG Navigator [2] - which accesses any data source with an OLE-DB provider and adds SQL capabilities to it if it does not have it. ISG Navigator is an OLE-DB provider itself and hence supports the standard OLE-DB interfaces. A data source such as a spreadsheet may have an OLE-DB provider but may not support SQL queries. Once "wrapped" with ISG Navigator, this source can be accessed through OLE-DB interfaces using SQL. An application can access this source even if it has no knowledge of spreadsheets. ISG Navigator as a wrapper is illustrated in Figure 3.2.

### 3.2.2 AURORA Mediators As Middleware Components

As described in Section 1.2.1 and shown in Figure 1.2, AURORA provides specialized mediators supporting flexible data models. Like other mediators, the AURORA mediators and wrappers can be composed to perform increasingly complicated data mediation. This section discusses various scenarios of such compositions.

In the AURORA context, when mediator $M_1$ *composes* with $M_2$, one of them will access the other to make use of the data the latter mediator is able to serve. However, AURORA mediators are specialized, the integration mediators are able to access multiple other mediators, and the homogenization mediators are able to deal with single sources. Generally, AURORA mediator composition



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

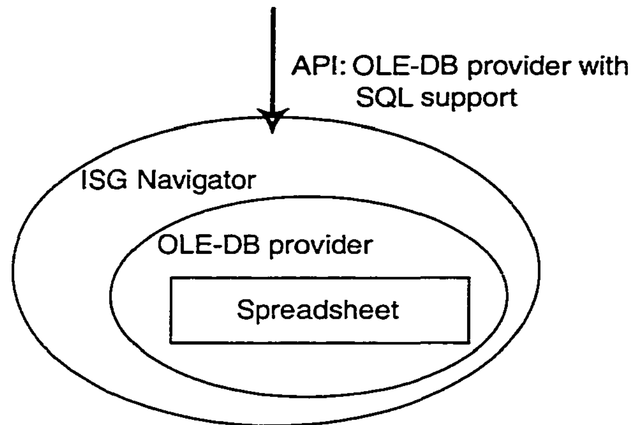



Figure 3.2: ISG Navigator as a Wrapper

must follow the following rules:

1. Integration mediators can only access homogenization mediators.

2. Homogenization mediators can access mediators of any kind, including wrappers, integration mediators, and other homogenization mediators.

3. Mediators that support an object data model should have the built-in capability of accessing those supporting the relational data model, but not vice versa.

A data mediation system can be constructed by using a network of mediators that cooperate with one another to provide an integrated data service. The use of AURORA mediators in building middleware is illustrated by Figures 3.3 and 3.4. The left of Figure 3.3 illustrates a scenario where AURORA mediators supporting the relational data model are used to construct a data mediation system that provides a relational service view. The diagram on the right of Figure 3.3 illustrates a scenario where all mediators support an object data model. Figure 3.4 illustrates how mediators supporting different data models can be composed. In this diagram, sources 1 and 2 are wrapped to support the relational data model. These two sources are first homogenized with respective relational homogenization mediators and then integrated with a relational integration mediator. Eventually, these sources participate in the object-oriented mediator at the top of Figure 3.4. To do this, the relational integration mediator - the left-most RI mediator - is treated as a relational source and is homogenized by an object-oriented homogenization mediator before it is composed with the target OI mediator on the top of the diagram.

Generally, a mediation scenario is determined by the data model of the service view and that of the data sources. Sources can be wrapped with a relational wrapper or an object-oriented wrapper, whichever is most conveniently built. There may be many types of data sources but after they



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

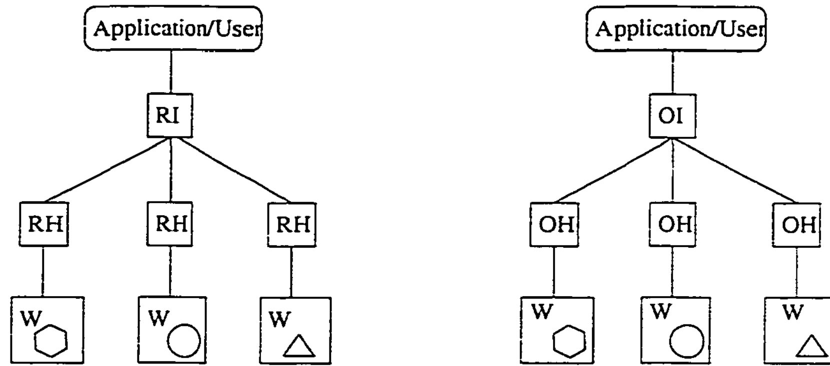



Figure 3.3: AURORA Application: uniform

are wrapped, there are only two types of sources as far as AURORA mediators are concerned: relational and object-oriented. The service view can be relational or object-oriented. Therefore, various mediation scenarios can be summarized in Table 3.1. Each entry in this table refers to a diagram that depicts an example composition of AURORA mediators supporting the corresponding mediation scenario. Most entries in this table have been explained in the previous paragraph. The N/A entries in this table represent scenarios that cannot be realized using AURORA mediators. These are scenarios where the service view is relational but one or more of the sources is object-oriented.

| Source Wrapper / Service View | Relational | Object-oriented | Mixed |
|---|---|---|---|
| Relational | Figure 3.3: left | N/A | N/A |
| Object-oriented | Figure 3.3: left | Figure 3.3: right | Figure 3.4 |

Table 3.1: AURORA's Flexible Data Model Support

### 3.2.3 Mediator Author's Toolkits (MATs) in AURORA

A general design principle of AURORA mediators is *"semi-automatic homogenization, automatic integration"*. The activity of homogenization deals with a wide range of semantic and structural mismatches between a source and a service view; this activity requires a mediator author to work with semantics. All AURORA homogenization mediators are equipped with a *Mediator Author's Toolkit* (MAT), which provides guidelines and facilities to a *mediator author*, performing homogeniza-



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

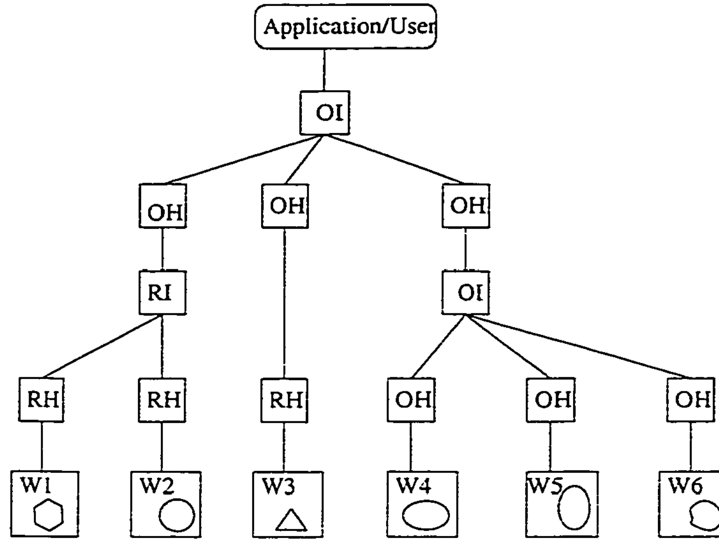



Figure 3.4: AURORA Application: mixed

tion. AURORA mediators with MATs are illustrated in Figure 3.5. A MAT provides a user-friendly interface that helps the mediator authors to perform homogenization systematically. It gathers various semantic knowledge from the mediator authors and stores it in an internal repository; this knowledge will be used by the homogenization mediator for query translation and processing. AURORA integration mediators deal with a small class of conflicts and are automatic, requiring no user interference in handling semantics; they do not have MATs attached.

## 3.3 Semantics of Integrated Data Services

Using the framework presented by [33], an integrated data service is provided as a *global database* whose schema is a service view in AURORA terms. This global database is virtual, storing no data. An integration mediator is in charge of accepting queries written in terms of this global database, translate it to queries against various sources, and assemble a query answer.

Semantics of well-known database query languages are defined based on *concrete databases*, databases that store data according to a schema. Generally, one can assume that, given any concrete database $D$ using a well-known data model, such as relational data model or the object-oriented data model, and a query, $Q$, written against the schema of $D$, in a well-known query language, such as SQL or OQL, that is compatible with the underlying data model of $D$, the answer to $Q$ using $D$, $answer(D, Q)$, has a well-understood meaning.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

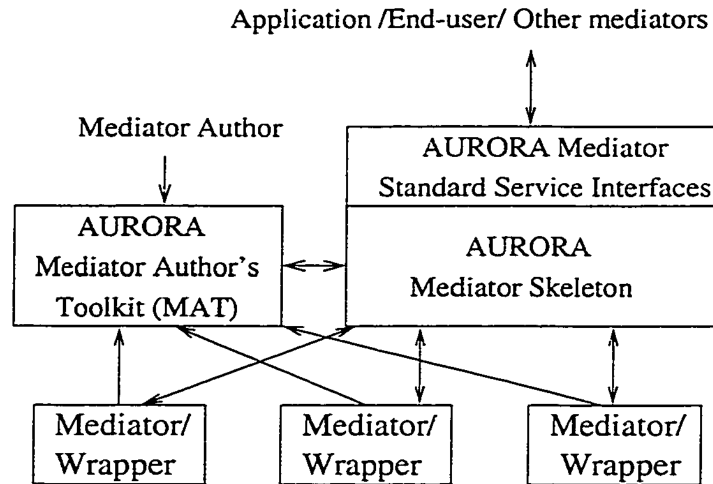



Figure 3.5: An AURORA Workbench

Since the global database in systems such as AURORA is virtual, semantics of queries posed against this database gives rise to the following question: ***what is the meaning of a query against a virtual global database?*** Although the global database is virtual, the participating sources are not virtual (when viewed as a data source). Fundamentally, this set of sources implicitly represent a global database. If one can construct the global database from the sources, then the semantics of queries posed against the virtual database is the same as those posed against any database where schema and content matches the constructed global database. Hence, before answering the question posed at the beginning of this paragraph, the following question must be answered: *Given a set of data sources, is there always one and only one global database? If so, what data are in it?* The rest of this section answers this question without using a formal model of semantics.

In AURORA, the answer to this question is affirmative. Each data source is homogenized by a mediator author, producing a homogenizing view. This view is derived procedurally by the mediator author using operators of her choice; it has well-defined semantics and can be materialized easily. Formally speaking, one can assume that for any given data source, $S$, there exists a function, $Hom_S$, that maps $S$ to a database instance. This instance, $Hom_S(S)$, is referred to as the *homogenized data source due to S*. In practice, function $Hom_S$ is constructed by a mediator author working to "hook up" $S$ to a chosen global database, using the AURORA homogenization facilities.

Once homogenization of all sources is completed, an integration mediator supporting the target service view also obtains a function *Fragments*, that maps each global relation name to one or more source relation names. That is, given any global relation name, $N$, $Fragments(N) = \{S_1.N_1, ...., S_k.n_k\}$, where for any $k \leq 0$, $N_k$ is the name of a relation in $Hom_{S_k}(S_k)$. In the rest of



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

this section, $HomRel(S.N)$ is used to denote the relation named $N$ in $Homs_S(S)$.

Given a service view $V_g$ containing relation definitions $R^g_1, ...., R^g_n$, and a set of source databases $S = \{S_1, ...., S_m\}$, the global database implicitly represented by $S$, with schema $V_g$, contains an instance for each relation defined in $V_g$. The instance of global relation $R^g_i$ is derived as follows:

$$inst(R^g_i) = MJ(PID(R^g_i), R^i_1, ...., R^i_x)$$

where $PID(R^g_i)$ is the plug-in identifier of global relation $R^g_i$, as specified in $V_g$, $R^i_j = HomRel(S^i_j.N^i_j)$, $Fragments(R^g_i) = \{S^i_1.N^i_1, ..., S^i_x.N^i_x\}$. For now, it is sufficient to think of the plug-in identifier as the primary key of the global relation. The concept of a fragment is described in more detail in Chapter 4. For the purposes of this section, a fragment can be considered as a *sound*, but not necessarily *complete*, view of a global relation, in terms of the framework described in [33].

The construction of a global database is deterministic, although it depends on the availability of two functions: *PID(R)*, which returns the plug-in identifier of global relation $R$, and $Fragments(R)$, which returns the set of source relations that are fragments of $R$. As described in Chapter 4, in the AURORA framework, $PID(R)$ must be provided as part of the service view definition, and the *Fragments* function is constructed automatically, upon completion of homogenization of all participating sources, by AURORA integration mediators that keep track of relationships between source relations and the target, global, relations. Intuitively, these two functions are made available by AURORA's mediation model. In other words, the mediation model of AURORA *requires* that the mediator authors provide these functions. In practice, the mediator authors are required to provide enough information so that these functions can be defined.

A few properties of the global database thus constructed are of interest, these include the following:

1. *Schema coverage*: Whether the global database schema can be derived from the source database schemas. On this property, it is assumed that the sources collectively provide a full coverage. Intuitively, this means that it is assumed that the sources collectively "have something to say about every domain of interest in the global schema".

2. *Entity coverage*: Whether the global database contains information on all entities of interest according to the semantics of the global schema. Currently, it is assumed that the sources collectively provide full entity coverage. This is a reasonable assumption. If the opposite is assumed, then one can assume that there is a "more complete" database, containing data on entities that can not be found in the known sources. These entities would not be of interest since their validity can not be verified.

3. *Data coverage*: Whether the global database contains "unknown" values and how these values are represented. This is a more complicated issue. In AURORA, it is assumed that data



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

coverage is partial and unknown values are represented as null values. Moreover, no atomic predicate evaluate to true if it is applied to a null value. A related issue is the problem of instance level conflicts in the global database. It is not clear whether these conflicts are considered and how in [33]. In AURORA, conflicts are retained in the global database, and the users query the database with the conflict tolerant querying facilities.

Once a global database is constructed, queries are answered using this database with well-known semantics. If the service view is relational, then the query semantics can be formally defined. If the service view is object-oriented, then the query semantics should conform to the standard chosen. In AURORA, OO query semantics should conform to the semantics of OQL queries as defined in the ODMG 2.0 standard [14]. Conflict tolerant querying in AURORA causes some extension to the well-known query semantics but these extensions are described in detail in Chapter 5.

As pointed out by [33], in systems such as Information Manifold [50], due to the type of data sources considered (sources may provide data that is irrelevant to the semantics of the global view), there may be infinite number of possible global databases. Hence the semantics of queries in these systems require new techniques to define and evaluate. In contrast, query semantics in AURORA raises fewer issues once the semantics of a global database is defined. The next paragraph describes AURORA in the terms of [33].

It is the mediator authors' responsibility to make sure that all the relations in homogenizing views are *sound* views of global relations, views that contains only tuples that fit into the global schema in terms of semantics. This means that all data sources are *open*, providing a true, although not necessarily complete, model of the world of interest. Consequently, the collection of data sources is always *consistent*. It is currently a conjecture that the query answers AURORA produces (modulo conflicts and CT querying) correspond to the *certain answers*, as described by [33], but this is yet to be proven. The other type of query answer, the *possible answer*, will be infinite and does not make sense.

## 3.4 Enabling Techniques in AURORA: A Roadmap

Each AURORA mediator requires a suite of enabling techniques. At the core of a mediator is a *Mediation Enabling Algebra* (MEA) that provides *Mediation Enabling Operators* (MEOs) that are suitable for manipulation of heterogeneous and autonomous data. MEAs must also be suitable for query processing in that they should facilitate optimized processing of mediator queries. Typically, a MEA consists of operators found in algebras that manipulate single-source data, such as relational algebra or object algebra, together with MEOs specially designed in AURORA for data manipulation required by homogenization or integration. The development of a MEA involves the following tasks:

1. Development of a mediator query rewriting algorithm to produce query evaluation plans



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(QEPs) in the MEA;

2. Development of query transformation rules in the MEA that potentially allow optimization of the above-generated QEPs;

3. Design of a query optimization strategy; and

4. Development of techniques for evaluating expensive MEOs efficiently.

Different mediators require different MEAs, depending on the type of mediation they perform and the data model they support. For homogenization mediators, a Mediator Author's Toolkit (MAT) must also be developed. This involves the design of a homogenization methodology, and a GUI-driven toolkit to support this methodology. The design of a MAT often proceeds that of a MEA, since it identifies the types of data manipulation required.

Technique suites are complete for AURORA-RH and AURORA-RI. The high-level design of a MAT for AURORA-OH is also complete. Complete development of MEAs for AURORA-OH and AURORA-OI are limited by the lack of a well-accepted object algebra as a starting point. However, the data manipulation operators in both of these mediators have been defined. How these operators form an algebraic framework to be used for query processing and optimization is a future research topic.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# Chapter 4

# Relational Mediation Framework

When a class of applications requires an integrated data service to be provided through a service view, based on a chosen data model and query language, a data mediation system needs to be constructed. Various tasks must be achieved in proper sequence in order to build such a system. A *mediation framework* defines these tasks, how they relate to one another, and how they are achieved. As discussed earlier, in AURORA, the data mediation process consists of two tasks: homogenization followed by integration. Therefore, the mediation framework of AURORA consists of two sub-frameworks: the homogenization framework and the integration framework. Moreover, when the service view data model is relational, the mediation framework works with relational data and is simpler than when the service view is in an object data model.

This chapter describes the relational mediation framework of AURORA in a bottom-up fashion, from how sources are to be wrapped, to how wrapped sources are homogenized and, finally, integrated. The topics covered include the following:

1. The homogenization framework and its realization by AURORA's Relational Homogenization mediator, AURORA-RH.

2. The integration framework and its realization by AURORA's Relational Integration mediator, AURORA-RI.

3. How the above frameworks and mediators work together to achieve data mediation.

## 4.1 An Overview of the Relational Mediation Framework

Construction of a data mediation system that supports a pre-defined service view based on a set of data sources (referred to as *participating sources*), starts with the activity of wrapping the sources. Wrapped sources are first homogenized to remove their idiosyncrasies with respect to the service view, and then integrated into the access scope of the service view.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

### 4.1.1 Service View

For applications, the service view is a relational schema that can be queried. For sources that provide data through this view, it is a pre-defined relational schema where each relation, called a *global relation*, specifies a group of attributes as its **plug-in identifier** (PID). The PID is semantically a relational key and is used for *object matching*, the process of identifying source tuples that describe the same application entity; these source tuples must be combined to form tuples in global relations. For instance, a service view may contain the following relation with PID *"ISBN"*:

*Books(ISBN, title, year, oprahClub, bestSeller, category, NYTreview, avgReview, price)*

This relation contains information on books, their ISBN number, title, year of publication, whether it is chosen by the Oprah's reading club, whether it is a national best seller, the rating by the New York Times, the average rating of customer reviews, and the price of the book. Intuitively, the PID is a "ticket" that a source tuple must produce in order to identify itself in the context of the global relation to which it contributes data. Tuples from different sources holding the same PID are considered to describe the same application entity, and are combined to form a tuple in the global relation.

In the rest of this chapter, the following notations are used. $PID(R)$ is used to denote the PID of global relation $R$. To simplify presentation, most of the time it is assumed that $PID(R)$ consists of a single attribute. For $t \in R$, its PID value is denoted as $t.PID$. For any global relation $R$, $R\{PID(R)\}$ denotes the set of PID values appearing in $R$.

### 4.1.2 Data Sources and Relational Wrappers

In order to participate in a service view supported by an AURORA mediator, a data source must be accessible through an API known to AURORA mediators. In the relational context, this API must allow access to schema information, submission and execution of SQL queries, and collection of query results in tabular form. A *wrapper* is used to provide such an API. AURORA employs "thin" wrappers, that is, these wrappers do not remove any differences between a source schema and the target service view in structure or semantics: although accessible, a wrapped source could still be different from the service view in many ways.

Wrapper technology is an active area of research [75, 80], although it is not a focus of research in AURORA. As described in Chapter 7, currently, AURORA wrappers are constructed using commercially available middleware systems. This approach allows a wide range of data sources to be wrapped, but it is not a generic solution: there are sources that cannot be wrapped. However, as wrapper technology progresses, AURORA would be able to gain access to these sources as needed.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

### 4.1.3 Homogenization of Data Sources

Once wrapped, a data source must be homogenized to conform in structure and semantics to the target service view. The process of homogenizing a source requires derivation of a *homogenizing view* on top of the (wrapped) source. Some or all relations in this view must be *fragments* of various global relations. For instance, a relation *SomeBooks(ISBN, title, authors, publisher)* is a fragment of a global relation *Books(ISBN, title, year, oprahClub, bestSeller, category, NYTreview, avgReview, price)*. Intuitively, a fragment of a global relation provides data on *some* attributes of *some* tuples in this relation. For instance, *SomeBooks* is able to provide data on the ISBN number and title of books. Another data source maybe able to provide data on the category of the books described by *SomeBooks*. Yet another data source may provide data on books that are unknown to *SomeBooks*.

Formally, a source relation $R_s$ *qualifies* to be a fragment of a global relation $R_g$ if $PID(R_g) \subseteq ATTR(R_s)$, that is, $R_s$ is able to produce the PID attributes required by $R_g$. Whether a source relation is really a fragment of a global relation is a decision to be made by mediator authors. AURORA's mediation model determines that source data must be transformed into fragments of global relations in order to be included in a service view. Upon completion of homogenization, all the data that a source willingly exports and that are relevant to the target service view must exist as (view) relations in the homogenizing view. Moreover, these view relations must be specified, by a mediator author, as fragments of relevant global relations; this "fragment-of" relationship between source relations and global relations must be available at the time the global relation is derived. Generally, a homogenizing view may contain relations that are not specified as fragments of any global relation; these relations are irrelevant to the service view in that they will not be identified or accessed as data contributors. The relationship between source relations in homogenizing views and the target service view is illustrated in Figure 4.1.

Homogenization is performed by a *mediator author* as follows. The mediator author compares the source schema and the target service view to decide which portion of the service view the source is able to contribute data to, and then designs the homogenizing view accordingly. This design requires good understanding of the service view and the source schema, and requires many semantic decisions to be made. After a homogenizing view is designed, it must be derived from the source view by a mediator author following a *homogenization methodology*. This methodology is designed to help a mediator author to manage the complexities of the homogenization process.

The AURORA-RH mediator assists the mediator author in deriving a homogenizing view systematically. It does not design the homogenizing view, neither does it determine how a homogenizing view is to be derived from the underlying source schema; rather, it provides two closely related facilities to the mediator author: a Mediation Enabling Algebra (MEA), MEA-RH, and a Mediator Author's Toolkit (MAT), MAT-RH. MEA-RH provides an algebraic framework suitable for use in homogenization; it supports operators in the usual relational algebra and operators specially de-



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

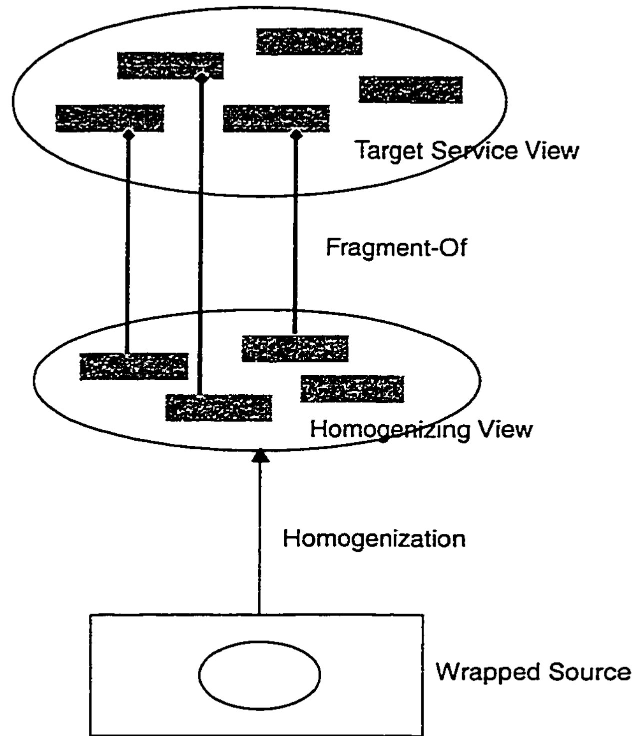



Figure 4.1: Homogenizing View and Service View

signed for homogenization. MAT-RH mandates the homogenization methodology that guides the mediator author to systematically identify and resolve structural and semantic differences between a source schema and the target service view. In each step of this methodology, certain types of derivations can be specified as an expression in MEA-RH. Section 4.2 describes how this methodology works and how MAT-RH supports it.

### 4.1.4 Integration of Homogenized Sources

Once homogenized, a data source should provide a description of the data it is able to contribute to the target service view. This description is constructed by the mediator author who specifies which source relation is a fragment of which global relation. Continuing with the previous example, if the data source wishes to contribute data towards *Books* through *SomeBooks*, it must make *SomeBooks* known to *Books* as a fragment. The integration framework must allow such a relationship to be specified and understood, and must also provide a mechanism for deriving global relations by combining all the known fragments.

In AURORA, integration is performed by AURORA-RI, the relational integration mediator. An AURORA-RI mediator supports a pre-defined service view by keeping track of fragments of the global relations, and using these fragments to derive global relations. The integration framework



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and the AURORA-RI mediator are described in Section 4.3.

## 4.2 AURORA-RH Homogenization Framework

The process of homogenization must remove all structural and semantic mismatches between a source schema and the target service view; this is a complicated process especially when multiple types of mismatches are present. In practice, not only do the mediator authors need constructs/operators to express resolutions of mismatches, they also need a *homogenization methodology* to ensure that identification as well as resolution of mismatches are performed systematically. AURORA-RH provides such a methodology and enforces it with a Mediator Author's Toolkit (MAT), called MAT-RH, that mandates a sequence in resolving mismatches of various types and provides facilities for expressing required resolutions. Intuitively, MAT-RH provides the mediator author with a skeleton of homogenization: the mediator author follows the homogenization methodology to systematically identify various mismatches and "hang" the resolutions of choice on the skeleton. MAT-RH maintains all the resolutions in an internal repository so that this knowledge can be used for the processing of mediator queries. The resolutions gathered by MAT-RH are expressed using the Mediation Enabling Algebra (MEA) of AURORA-RH, MEA-RH.

MEA-RH extends the relational algebra with operators specially designed for constructing homogenizing views. The extensions are to support more powerful *structural mapping* and *value mapping*. A structural mapping is a transformation that removes a difference in structure between the homogenizing view and the source schema, while a value mapping is a transformation that removes a difference in data values between the two. For instance, a relation in the homogenizing view may have an attribute whose values correspond to relation or attribute names in the underlying database. This is referred to as a *cross-over schema mismatch* [41]. It has been argued that the relational algebra cannot express a mapping that resolves this structural difference [45]. In contrast, MEA-RH can express such structural mappings. Defining homogenizing views often requires that arbitrary functions/look-up tables to be used to derive data values from the underlying database. Such value mappings may be allowed when defining relational views, but the characteristics of the mappings are not taken into consideration during processing of relational view queries.

### 4.2.1 The Homogenization Problem

This section gives a formal description of the homogenization problem and describes an example used for illustrating the homogenization process, and for demonstrating how facilities provided by AURORA-RH can be used by a mediator author to achieve homogenization.

Let $B$ be a relational database. Let $H$ be a homogenizing view consisting of relations $M_1, ..., M_n$. The problem of *homogenizing database $B$ into $H$* is to specify procedures, $P_i(B)(1 \leq i \leq n)$, that



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

construct relations $M_i$ $(i = 1, n)$ from the relations in $B$. $B$ is the *source database*; relations in $B$ are *source relations*; $M_i$ $(i = 1, n)$ are *target relations*. $H$ is also referred to as the *target view*.

| *- Source Database Schema -* |
|---|
| *Sales(month, hardcover, paperback, audiobook)*<br>*Travel(ISBN, title, price, deduction, bestSeller)*<br>*NewAge(ISBN, title, price, deduction, bestSeller)*<br>*Computer(ISBN, title, price, deduction, bestSeller)*<br>*Hobbies(ISBN, title, price, deduction, bestSeller)*<br>*Children(ISBN, title, price, deduction, bestSeller)* |
| *- Target View -* |
| *BookSales(month, book_type, salesAmt)*<br>*Books(ISBN, title, category, price, bestSeller)* |

Figure 4.2: A Homogenization Example

**Example 4.2.1** [ A Homogenization Example.] Figure 4.2 depicts a homogenization problem. The target view contains two relations: *BookSales*, which summarizes sales of various types of books, and *Books*, which describes all the books available, their ISBN number, title, category, price, and whether they are a national best seller. The source database provides similar information but is organized differently. Data on the monthly sales of different types of books are stored in relation *Sales*, which has one tuple for each month, with one column recording the sales of a particular type of books. Books in the same category are stored in a relation named after this category. In addition to these differences in structure, the following differences in semantics also exist: 1) in the source database, the sales and price data are recorded in Canadian dollars, while in the target view, the same data are to be reported in US dollars; 2) In the target view, *Books.price* is the cost of a book after deductions, while in the source database, the price is given as the regular price and a deduction rate; and 3) The target view perceives the "categories of books" differently from the source database. Rather than the categories of

$\{Travel, NewAge, Computer, Hobbies, Children\}$

the target view assumes that books are from the following categories:

{*Travel and Adventure, Alternative, Computer and Internet, Hobbies, Young Reader*}

The content of the source tables are shown in Figure 4.3. □

### 4.2.2 Mismatches and Resolutions

Each database defines *domains* that model *conceptual territories*. A domain is characterized by the following:



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

| Travel | | | | |
|---|---|---|---|---|
| ISBN | title | price | deduction | bestSeller |
| 001 | "Florida" | 45 | 0.15 | No |
| 002 | "China" | 67 | 0 | No |

| NewAge | | | | |
|---|---|---|---|---|
| ISBN | title | price | deduction | bestSeller |
| 003 | "Meditation" | 24 | 0 | No |
| 004 | "Dreams" | 23 | 0.20 | Yes |

| Computer | | | | |
|---|---|---|---|---|
| ISBN | title | price | deduction | bestSeller |
| 005 | "TCP/IP" | 41 | 0.15 | Yes |
| 006 | "HTML" | 37 | 0.20 | Yes |

| Hobbies | | | | |
|---|---|---|---|---|
| ISBN | title | price | deduction | bestSeller |
| 007 | "Pens" | 74 | 0.15 | No |
| 008 | "Quilts" | 45 | 0.30 | No |

| Children | | | | |
|---|---|---|---|---|
| ISBN | title | price | deduction | bestSeller |
| 009 | "Micky" | 10 | 0 | No |
| 010 | "Pooh" | 8 | 0 | No |

| Sales | | | | |
|---|---|---|---|---|
| month | hardcover | paperback | audiobook | |
| Feb/99 | 6700 | 6900 | 800 | |
| Mar/99 | 7600 | 8400 | 7800 | |

Figure 4.3: Source Tables

1. Its conceptual territory.
2. Its representation construct in the relational data model, whether it is represented as relations, attributes, or data values.
3. The data type and semantics of its elements.

For instance, the conceptual territory of "title of books" is modeled by domain *Books.title* in the target view; the elements of this domain are data values of the attribute *title* of relation *Books*; and these elements are character strings. A domain can be a *meta domain*, consisting of relations and attributes, or a *data domain*, consisting of values in a relation. For example, the conceptual territory of "book categories" is modeled by a meta domain $\{Travel, NewAge, Computer, Hobbies, Children\}$ in the source database in Figure 4.2. The representation construct of this domain is relation, the elements of this domain are relation names. Domains from different databases that model the same conceptual territory are said to be *corresponding domains*. When corresponding domains are different in their representation constructs, data types or semantics of elements, there is a *domain mismatch*.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Consider a source database $B$ and a target relation $M$ to be derived from $B$. Fundamentally, deriving $M$ from $B$ requires deriving the domains of each attribute of $M$ from $B$ and then combining these domains together to form relation $M$. Each attribute $A$ of $M$ defines a data domain $D_A^M$ that models a conceptual territory $C_A$, which is also modeled by $B$. If $B$ models $C_A$ with the same representation construct, data type, and semantics as $M.A$, then derivation of the domain of $M.A$ is easy. If not, that is, there are domain mismatches over conceptual domain $C_A$, deriving $M.A$ from $B$ requires removal of these domain mismatches.

Generally speaking, given a source database $B$ and a data domain $D$ that models a conceptual territory $C_A$, the following types of domain mismatches between $D$ and its corresponding domains in $B$ may arise:

**Type 1 cross-over schema mismatch.** A type 1 cross-over schema mismatch happens when $C_A$ is modeled by a domain consisting of relation names in $B$.

**Type 2 cross-over schema mismatch.** A type 2 cross-over schema mismatch happens when $C_A$ is modeled by a domain consisting of attribute names in $B$.

**Domain structural mismatches.** A domain structural mismatch happens when $C_A$ is modeled by more than one domain(s) in $B$.

**Domain element mismatches.** A domain element mismatch happens when $C_A$ is modeled in $B$ by a domain whose elements are of a different data type or semantics from the elements of $D$.

According to the definition of a domain given earlier, the above list covers all possible cases of mismatches between a target data domain $D$ and its corresponding domains in a given database. These mismatches are illustrated by the following example.

**Example 4.2.2** The example shown in Figure 4.2 demonstrates the following mismatches:

1. Type 1 cross-over schema mismatch: In the target database, the concept of "book categories" is represented as data domain *Books.category*. The same concept is represented as relation names in the source database.

2. Type 2 cross-over schema mismatch: In the target database, the concept of "type of books" is represented as data domain *BookSales.book_type*, but is represented as attribute names in the source database.

3. Domain structural mismatch: In the target database, *Books.price* means the price including deductions. In the source database, the same concept is represented by two data domains: *Price* = *Travel.price* ∪ *NewAge.price* ∪... *Children.price* and *Deductions* = *Travel.deduction* ∪ *NewAge.deduction* ∪... *Children.deduction*.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4. Domain element mismatch: In the target database, the domain *Books.price* contains values that represent money amounts in US dollars while, in the source database, the domain *Price* as described earlier contains values that represent amounts in Canadian dollars.

5. Domain element mismatch: In the source database, the domain representing the concept of "book categories" contains elements whose values are from the collection of strings {*Travel, NewAge, Computer, Hobbies, Children* }. In the target view, elements of the domain that represents the same concept, *Book.category*, draw their values from {*Travel and Adventure, Alternative, Computer and Internet, Hobbies, Young Reader*}.

□

A mismatch is resolved by deriving a view in which this particular mismatch no longer exists. However, the derived view may contain other mismatches that require more views to be derived in order to remove them. Therefore, such view derivations can be done as many times as it takes until all mismatches are resolved. Each derivation aims at solving particular mismatches and may require special transformation to the data. These transformations are expressed by the usual relational operators as well as the AURORA *primitives*, operators specially designed for resolving the mismatches described above.

### 4.2.3 AURORA-RH Primitives

AURORA-RH primitives are Mediation Enabling Operators (MEOs) specially designed to facilitate homogenization. These primitives consist an extension to the relational algebra to form MEA-RH, a Mediation Enabling Algebra (MEA) that is the basis for performing homogenization and, later, for processing queries. All primitives take a relation as an argument and generate a relation; they compose with relational operators in a well-defined manner.

In this dissertation, $ATTR(R)$ denotes the set of attributes in relation $R$, $RELname(R)$ denotes the name of relation $R$, and $ATTRname(A)$ denotes the name of attribute $A$. Let $B$ be the source database to be homogenized. AURORA-RH provides the following primitives:

**primitive** *retrieve.*

Let $Q$ be an expression in relational algebra over the source relations in database $B$,

$$R' = retrieve(Q)$$

submits query $Q$ to database $B$ and returns the result table $R'$.

**primitive** *pad.*

Let $R$ be a relation, $A$ be an attribute, $A \notin ATTR(R)$, and $c$ a constant,



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$R' = pad(R, A, c)$$

defines a relation $R'$, $ATTR(R') = ATTR(R) \cup \{A\}$. The population of $R'$ is defined by

$$R' = \{t' \mid t'[A] = c;\ t'[A'] = t[A'], t \in R, A' \in ATTR(R)\}$$

Intuitively, for each tuple $t \in R$, *pad* generates a $R'$ tuple $t'$ by "padding" $t$ with a new field $A$ with value $c$. *pad* is useful for restructuring relations. Consider the relation *Travel* in Figure 4.2. Let $R' = pad(\ retrieve(Travel),\ category,\ \text{"Travel"})$. $R'$ has scheme *(ISBN, title, price, deduction, category)* and a population consisting of all the *Travel* tuples each *tagged* with relation name *"Travel"* as attribute *category*.

**primitive** *rename.*

Let $R$ be a relation, $A \in ATTR(R)$, and $n$ be an attribute name, such that no attribute in $R$ has name $n$, then

$$R' = rename(R, A, n)$$

defines a relation $R'$ with scheme identical to the scheme of $R$ with attribute $A$ renamed to $n$. The population of $R'$ is defined by the following:

$$R' = \{t' \mid t'[n] = t[A], t'[A'] = t[A'], t \in R, A' \in ATTR(R) - \{A\}\}$$

**primitive** *deriveAttr.*

Let $R$ be a relation. Let $L_i \subseteq ATTR(R)(i = 1, k)$ be a list of attributes in $R$. Let $N_i(i = 1, k)$ be attributes. Let $f_i$ be functions of appropriate signatures.

$$R' = deriveAttr(R, L_1, N_1, f_1, ...., L_k, N_k, f_k)$$

defines a relation $R'$, $ATTR(R') = ATTR(R) \cup \{N_1, ..., N_k\}$. The population of $R'$ is defined by:

$$R' = \{t' \mid t'[N_i] = f_i(t[L_i]), 1 \leq i \leq k;\ t'[A] = t[A], A \in ATTR(R) - \{N_1, ..., N_k\}. t \in R\}$$

Intuitively, for each tuple $t$ of $R$, *deriveAttr* generates a tuple $t'$ of $R'$ by adding fields $N_i$ to $t$ $(i = 1, k)$ and sets their values to be $f_i(t[L_i])$, where $t[L_i]$ is the list of values obtained by projecting $t$ over $L_i$. If an attribute in $R$ has the same name with some $N_s$ $(1 \leq s \leq k)$, this attribute is replaced by $N_s$.

*deriveAttr* is used for resolving domain mismatches with arbitrary functions, as shown in Sections 4.2.5 and 4.2.5. Notice that functions $f_i$ in *deriveAttr* are not aggregates; they apply to field(s) in a single tuple, while aggregates apply to multiple tuples. Given a table containing student grades, *deriveAttr* cannot be used to derive an attribute "*GradeAverage*"; it can be used to derive the basic student-grade table. "*GradeAverage*" can then be derived using the usual aggregates.

A **transformation expression** is an expression in MEA-RH that defines the derivation of the scheme and population of a relation from given relations and other arguments. A transformation expression deriving relation $R$ is in the form of $R = T_E$. If $T_E$ is in the form of $retrieve(Q)$, where $Q$ is a relational algebra expression, $R$ is a *direct relation*; otherwise, if $T_E$ is an expression that



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

involves other MEOs, such as *pad*, *rename*, or *deriveAttr*, $R$ is a *derived relation*. Intuitively, a direct relation is the immediate result of a query over the source database.

### 4.2.4 Homogenization Methodology and AURORA-RH

When deriving a target homogenizing view $H$ from a source database $B$, multiple domain mismatches are often encountered. For instance, over the conceptual territory of "book categories", there are two domain mismatches: a type 1 cross-over mismatch and a domain element mismatch (mismatches 1 and 5 in Example 4.2.2). A *mediation methodology* mandates a sequence in which these mismatches should be identified and resolved. Such methodologies are designed to assist the mediator author in examining and resolving mismatches systematically and is a pragmatic means for making the process of homogenization more manageable. Many different methodologies can be invented. One such methodology, called the *homogenization methodology*, is designed as part of the AURORA homogenization framework. This methodology mandates that homogenization be performed in the following 6 steps:

1. Schema import;
2. Resolve type 1 schema mismatches;
3. Resolve type 2 schema mismatches;
4. Link relations;
5. Resolve domain structural mismatches; and,
6. Resolve domain unit/population mismatches.

In step 1, the schema import step, the mediator author selects the portion of the source database $B$ that is relevant to the target view $H$. It is possible that the whole schema of $B$ is relevant. More often, only some portion of some relations of $B$ are of interest. Major structural differences are eliminated in steps 2 and 3; after step 3, all domain mismatches that are left are between data domains of the source and those of the target view. In step 4, the relation-linking step, the mediator author must gather relevant domains in a meaningful way to derive one distinguished relation for each target relation $M$ in $H$. This relation, called the *prototype relation* of $M$, should contain all the data domains that correspond to data domains in $M$. Finally, in steps 5 and 6, the mediator author specifies how data domains in each target relation are to be derived, by resolving mismatches between these data domains and their corresponding domains in the prototype relation. These mismatches are often resolved using user-defined functions and look-up tables.

Each step of the homogenization methodology is characterized by the following:

1. Input: In each step, certain relations should be examined.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. Operators: MEOs and transformations that can be used in this step.

3. Output: domain mappings and other semantic information, such as enumerated domains, look-up tables, and derived relations, that may be generated by this step.

AURORA's homogenization methodology is enforced by a Mediator Author's Toolkit in AURORA-RH, called MAT-RH. Each step of the methodology is supported by a specialized tool. MAT-RH mandates that tools be invoked in sequence. The tools are designed to provide the following facilities:

1. Restrict the scope of the supported mediation step. For instance, steps after relation linking should only work with the prototype relations. Tools supporting these steps should ensure that no other relations are visible or manipulated.

2. A user-friendly interface to assist the mediator authors in specifying transformations and other information used for view derivation.

3. Transformations specially designed for resolving complicated mismatches. For instance, a transformation for resolving type 1 cross-over mismatch, *RELmat*, is provided by SME-1, the tool supporting resolution of type 1 cross-over mismatches, which is step 2 of the methodology. This transformation is not available in any other tools.

4. Store the transformations, domain mappings and other semantic information provided by the mediator author for homogenization in an internal repository, in a proper format, so that this information can be used later for query processing.

The architecture of AURORA-RH is shown in Figure 4.4. **MAT-RH** consists of 6 tools, named IE, SME-1, ..., DEE, that support the 6 steps of the homogenization methodology, respectively. These tools will be described later in this chapter. Each tool gathers and stores various semantic information into the **View Definition Repository**. **AURORA-RH Primitives** implements primitives described in Section 4.2.3. **AURORA-RH Query Processor (AQP)** processes queries posed against the target view. It translates such a query into a set of queries over the source database, using the mapping information from the View Definition Repository, sends these queries for execution and assembles the final answer from the returned data, using the primitives. The query processing techniques of AURORA-RH is described in Chapter 5.

### 4.2.5 Homogenization with AURORA-RH

This section demonstrates the homogenization methodology by walking through the process of deriving the target view from the source database, as given in Figure 4.2. As each step is performed, the tool in MAT-RH supporting this step will be described.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

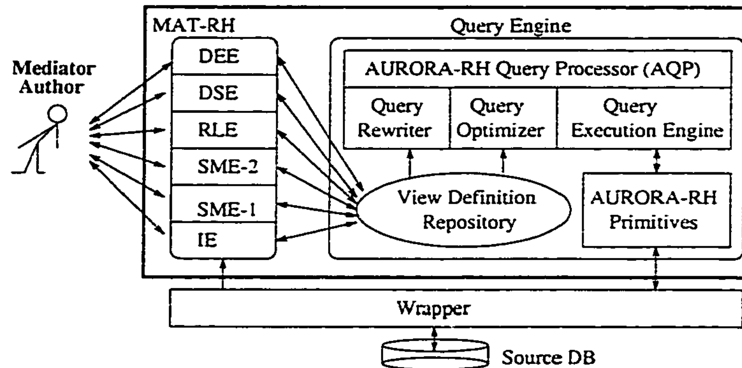



Figure 4.4: Architecture of AURORA-RH Workbench

**Step 1: Schema Import Using the Import Environment (IE)**

The input to step 1 includes all the source relations exported by a source database $B$. In this step, the mediator author can derive relations using relational algebraic expressions over $B$; these expressions select portions of $B$ that are relevant to the target view. The output of this step is a relational schema, referred to as the *import schema*, that contains a set of direct relations of the form of $R = retrieve(Q)$, where $Q$ is a relational algebraic expression over database $B$.

**Example 4.2.3** [Importing source database.] In the example given in Figure 4.2. all relations are of interest and hence are imported in full. The schema import step produces a set of direct relations $R = retrieve(R)$, where $R \in$ { *Travel, NewAge, Computer, Hobbies, Children*}. The repetition of relation names on the two sides of the *retrieve* operator causes no confusion because any relation name referenced by the parameter of *retrieve* refers to source relations, and the relations on the left hand side are always view relations. Generally, relations in the imported schema can be derived by expressions such as $Adult = retrieve(\sigma_{Age>18} SomePeople)$, when the target view is used to provide data on adults only. □

The step of schema import is supported by the Import Environment (IE) tool. The input to IE includes all the source relations exported by a source database $B$. The main facilities provided by IE is a source schema browser, which displays the structure of the source schema. IE also supports common schema import options, such as importing the entire source schema. In this case, derivation of view relations are automatically generated by IE so that the mediator author does not have to write derivations such as $R = retrieve(R)$ for each relation.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**Step 2: Solving Type 1 Cross-over Schema Mismatches and SME-1**

Step 2 of the homogenization methodology requires the mediator author to remove type 1 cross-over schema mismatches. The input to this step includes all the relations in the imported schema produced by step 1. All relational operators and AURORA primitives can be used for deriving view relations in this step. A special transformation, *RELmat*, is also available.

Given $D^R = \{R_1, ..., R_n\}$, a group of relations with identical schemes, let $A$ be an attribute, $A \notin ATTR(R_1)$, then:

$$RELmat(D^R, A) = \bigcup_{i=1}^{n} pad(R_i, A, RELname(R_i))$$

The result relation has attribute set $ATTR(R_1) \cup \{A\}$. The population of the result relation contains tuples from all the relations in $D^R$, each tagged with a new field $A$ that contains the name of the relation it came from. For example, if $D^R = \{$ *Travel, NewAge, Computer, Hobbies, Children* $\}$. and if the relations in $D^R$ contain tuples as shown in Figure 4.3, then the following transformation

$$Books_p = RELmat(D^R, category)$$

derives relation $Books_p$, as shown in Figure 4.5. *RELmat* transforms a meta domain, the relation group, into a data domain. That is, the relation derived with *RELmat* contains a data domain whose elements are relation names. For instance, the relation shown in Figure 4.5 contains a data domain *category* that draws its values from a set of relation names, {*Travel, NewAge, Computer, Hobbies, Children*}. Step2 of the homogenization example is given in the following example.

| ISBN | title | price | deduction | bestSeller | category |
|---|---|---|---|---|---|
| 001 | "Florida" | 45 | 0.15 | No | "Travel" |
| 002 | "China" | 67 | 0 | No | "Travel" |
| 003 | "Meditation" | 24 | 0 | No | "NewAge" |
| 004 | "Dreams" | 23 | 0.20 | Yes | "NewAge" |
| 005 | "TCP/IP" | 41 | 0.15 | Yes | "Computer" |
| 006 | "HTML" | 37 | 0.20 | Yes | "Computer" |
| 007 | "Pens" | 74 | 0.15 | No | "Hobbies" |
| 008 | "Quilts" | 45 | 0.30 | No | "Hobbies" |
| 009 | "Micky" | 10 | 0 | No | "Children" |
| 010 | "Pooh" | 8 | 0 | No | "Children" |

Figure 4.5: $Books_p$: Result of RELmat

**Example 4.2.4** [ Solving type 1 cross-over schema mismatch.] The source database models the concept of "book category" as relation names, while the target view models it as a data domain *Books.category*. The following is the resolution to this mismatch, using the transformation *RELmat*:

$D^R = \{Travel, NewAge, Computer, Hobbies, Children\}$



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$Books_p = RELmat(D^R, category)$

As shown in Figure 4.5, relation $Books_p$ has scheme *$Books_p$(ISBN, title, price, deductions, bestSeller, category)*. The data domain *category* contains elements whose values are from $D^R$. □

The output of this step of the homogenization methodology consists of all the relations produced in the previous step, as well as those derived in this step. At the end of this step, the result view should contain no type 1 cross-over schema mismatch.

The tool of Schema Mismatch Environment 1, SME-1, in MAT-RH supports this step. The main facility provided by SME-1 is a template for constructing a *RELmat* transformation, so that the mediator author does not have to write a formula, but rather fills out a form that is designed to collect various information needed for a *RELmat* transformation. The transformation itself is generated by SME-1.

### Step 3: Solving Type 2 Cross-over Schema Mismatches using SME-2

Step 3 of the homogenization methodology requires removal of type 2 cross-over schema mismatches. The input to this step includes all the relations in the output of step 2. All relational operators and AURORA primitives can be used for deriving view relations in this step. A special transformation, *ATTRmat*, is also available.

*ATTRmat*(attribute materialize) is a special transformation used for resolving type 2 cross-over schema mismatches. Given $D^A = \{A_1, ..., A_n\}$, a group of attributes in a relation $S$ that have identical data types, let $N_A$ and $N_V$ be attribute names, $N_A, N_V \notin ATTR(S)$, then:

$ATTRmat(S, D^A, N_A, N_V)$

$= \bigcup_{i=1}^{n} pad(rename(\pi_{ATTR(S)-D^A\cup\{A_i\}}(S), A_i, ATTRname(N_V)), N_A, ATTRname(A_i))$

The result relation has attribute set $ATTR(R) - D^A \cup \{N_A, N_V\}$. Attribute $N_A$ is the name of the attribute in the derived relation whose domain corresponds to $D^A$. Attribute $N_V$ is the name of the attribute in the derived relations whose domain corresponds to the domain of the attributes in $D^A$. For instance, if $D^A$ = { *hardcover, paperback, audiobook* } and relation *Sales* contains tuples as shown in the lower right corner of Figure 4.3, then the following transformation

$$Sales_p = ATTRmat(Sales, D^A, book_type, salesAmt)$$

derives a relation $Sales_p$, as shown in Figure 4.6. *ATTRmat* transforms a meta domain, the attribute group, into a data domain. For instance, the table shown in Figure 4.6 contains a data domain "book_type", whose elements draw their values from $D^A$. Step 3 of the homogenization methodology, and the application of *ATTRmat*, are illustrated by the following example.

**Example 4.2.5** [Solving type 2 schema mismatch.] The target view models the concept of "types of books" as a data domain *BookSales.book_type*, while the source database models it as attributes



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

| month | salesAmt | book_type |
|---|---|---|
| Feb/99 | 6700 | "hardcover" |
| Mar/99 | 7600 | "hardcover" |
| Feb/99 | 6900 | "paperback" |
| Mar/99 | 8400 | "paperback" |
| Feb/99 | 8000 | "audiobook" |
| Mar/99 | 7800 | "audiobook" |

Figure 4.6: $Sales_p$: Result of ATTRmat

*hardcover*, *paperback*, and *audiobook* in relation *Sales*. The following is the resolution for this mismatch:

$D^A = \{hardcover, paperback, audiobook\}$

$BookSales_p = ATTRmat(Sales, D^A, book_type, salesAmt)$

As shown in Figure 4.6, relation $BookSales_p$ has scheme *(month, salesAmt, book_type)*. The data domain *book_type* contains elements that draw their values from $D^A$. □

The output of this step of the methodology consists of all the relations produced by the previous step, as well as those derived in this step. At the end of this step, the result view should contain no type 1 or type 2 cross-over schema mismatch.

The tool of Schema Mismatch Environment 2 of MAT-RH, SME-2, supports this step. The main facility provided by SME-2 is a template for constructing an *ATTRmat* transformation, so that the mediator author does not have to write a formula, but rather fills out a form that is designed to collect various information needed for an *ATTRmat* transformation. The transformation itself is generated by SME-2.

### Step 4: Relation Linking and RLE

Assume that a target relation $M$ has attribute $A_1, ..., A_n$, modeling conceptual territories $C_1, ..., C_n$. After both types of cross-over schema mismatches are removed by steps 2-3, $C_1, ..., C_n$ would now be modeled by data domains in the output view of Step 3. The step of relation linking requires the mediator author to combine all the data domains that are related to $C_1, ..., C_n$ to form a distinguished relation $M_p$. $M_p$ contains all the data domains modeling $C_1, ..., C_n$ and is called a *prototype* of $M$. Moreover, attribute names in $M_p$ must satisfy the following condition: for any attribute $A'_i \in ATTR(M_p)$, if it models the conceptual territory of $C_i$, then $A'_i$ should have the same attribute name as $A_i \in ATTR(M)$. Intuitively, this condition ensures that if an attribute $A'$ of $M_p$ is "the same" as attribute $A$ of target relation $M$, it should bear the same name as the latter. The output of the relation linking step is a set of prototype relations, one for each of the target relations.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**Example 4.2.6** [Relation linking. ] In Examples 4.2.4 and 4.2.5, relations $Books_p$ and $BookSales_p$ are defined. These two relations are the prototypes of target relations *Books* and *BookSales*, respectively, and they are the output of the Relation Linking step. No derivation is explicitly performed in the relation linking step in this example but in general, one or more view relations can be derived to facilitate the final derivation of the prototype relations □

MAT-RH supports the step of relation linking with the tool of Relation Linking Environment. RLE. The main facility provided by RLE is for the mediator authors to derive and mark the distinguished relations as the only output relations.

### Step 5: Solving Domain Structural Mismatches using DSE

Step 5 of the homogenization methodology requires the mediator authors to remove domain structural mismatches between the prototype relations, which are the output of the relation linking step, and the respective target relations.

Consider a target relation $M$ and an attribute $A \in ATTR(M)$ that models a conceptual territory $C_A$. After the relation linking step, $C_A$ might be modeled by the prototype relation of $M$, $M_p$, as one data domain or as more than one data domain. Step 5 of the homogenization methodology requires that for each attribute $A_i$ of $M$ that corresponds to more than one data domain in $M_p$, the mediator author specify the following:

1. $L_i = \{A^S_{i,1}, ..., A^S_{i,l_i}\}$, attributes in $M_p$ that correspond to $A_i$. $L_i$ is referred to as the *source domain list* of attribute $A_i$.

2. A *domain structural function (DSF)*, $f^s_i$ with the following signature:

$$f^s_i : T^S_{i,1} \times ... \times A^S_{i,l_i} \rightarrow T_i$$

   where $T^S_{i,l} (1 \leq l \leq l_i)$ is the data type of attribute $A^S_{i,l}$ of $M_p$, and $T_i$ is the data type of attribute $A_i$ of relation $M$.

DSFs are arbitrary functions that must be provided by the mediator author. Once all the DSFs are specified, the mediator author can derive a relation $M_v$ as follows:

$$M_v = \pi_{A_1,...,A_m}(deriveAttr(M_p, L_{d_1}, A_{d_1}, f^s_{d_1}, ...., L_{d_k}, A_{d_k}, f^s_{d_k}))$$

where $A_{d_1}, ..., A_{d_k} (0 \leq k \leq m, 1 \leq d_i \leq m)$ are all the attributes of $M$ that correspond to more than one attribute of $M_p$. The schema of relation $M_v$ is similar to that of the target relation $M$ except that an attribute $M_v.A$ may have a different data type or meaning, such as unit of measure, from attribute $M.A$. $M_v$ is referred to as the *value model* of relation $M$ since the only difference between $M_v$ and $M$ is in the data values they contain; these differences are due to domain element mismatches which are to be removed in the next step of homogenization.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**Example 4.2.7** [ Solving domain structural mismatch.] In relation $Books_p$ derived in Example 4.2.4, attributes *price* and *deduction* together describe the over-the-counter price of a book. In target relation *Books*, *price* means over-the-counter price, including all deductions. To resolve this mismatch, the following is specified:

$L_{Books.price} = \{price, deduction\}$

$f^s_{Books.price}(p, d) = (1 - d) * p$

Relations $Books_v$ can be derived as follows:

$$Books_v = \pi_{ISBN,title,category,bestSeller,price}(deriveAttr(Books_p, \{price, deduction\}, price, f^s_{Books.price}))$$

There is no domain structural mismatch over the domains of relation *BookSales*. Hence

$$BookSales_v = BookSales_p$$

□

The output of this step of the methodology includes the following:

1. Source domain lists and the DSFs for all the attributes of $M$ that correspond to more than one domain in the prototype relation;

2. The derivation of the value model relation for each target relation.

MAT-RH supports this step with the Domain Structure Environment, DSE, which supports the following:

1. It provides templates for the mediator authors to specify the source domain lists and the DSF for attributes of $M$.

2. It ensures that the DSFs are provided with the appropriate signatures.

3. It automatically creates the derivation of the value model relations, using the source domain lists and the DSFs specified by the mediator author.

**Step 6: Solving Domain Element Mismatches and DEE**

The value model relation of a target relation $M$, $M_v$, as derived by step 5 of the methodology, has a scheme similar to that of $M$. However, for an attribute $A \in ATTR(M)$, the values of $M_v.A$ may differ from that of $M.A$ in data type and/or in semantics. For instance, they might be based on different units of measurement. To derive $M.A$ from $M_v.A$, step 6 of the homogenization methodology requires the mediator to specify a ***domain value mapping*** that converts values of $M_v.A$ to that of $M.A$ when needed. If a domain value mapping maps each $M_v.A$ value to a unique $M.A$ value, it is a ***domain value function (DVF)***. Otherwise, there is ***uncertainty*** in the homogenization



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

process. In this dissertation, only DVFs are considered. Inverses of DVFs, if they exist, must also be specified; they are used for efficient query processing, as described in Chapter 5. After the DVFs are specified, the target relation can be derived with the operator *deriveAttr* as follows:

$$M = deriveAttr(M_v, \{A_1\}, A_1, f_1^v, ...., \{A_k\}, A_k, f_m^v)$$

where $ATTR(M) = \{A_1, ..., A_m\}$ and $f_i^v(i = 1, m)$ is the DVF for attribute $A_i$. This step of the mediation methodology is illustrated by the following examples.

**Example 4.2.8** [ Solving domain element mismatch.] Consider relation $Books_v$ in Example 4.2.7. The semantics of attribute *price* is "price including deduction" but the price is still in Canadian dollars. In the target view, the price is represented in US dollars. Assume 1 US dollar is worth 1.5 Canadian dollars, then the DVF for *Books.price* can be defined as:

$$f^v_{Books.price}(s) = CDNtoUSD(s) = s/1.5$$

Similarly, the DVF for *BookSales.salesAmt* can be defined as

$$f^v_{BookSales.salesAmt}(s) = CDNtoUSD(s)$$

$CDNtoUSD()$ has an inverse that should be specified by the mediator author as well. □

**Example 4.2.9** [ Solving domain element mismatch.] Consider relation $Books_v$ in Example 4.2.7. The domain of $Books_v$*.category* consists of values from {*Travel, NewAge, Computer, Hobbies, Children*}, while domain *Books.category* consists of values from {*Travel and Adventure, Alternatives, Computer and Internet, Hobbies, Young Readers*}. To resolve this mismatch, a DVF must be specified for *Books.category*. This DVF is given as a mapping table *categoryMap* shown in Table 4.1. That is, $f^v_{Books.category}(j) = categoryMap(j)$. This mapping is 1-1 and is invertible.

Combining all the DVFs specified above with those specified in the previous example, the relations *Books* and *BookSales* can be derived as follows:

$Books = deriveAttr(Books_v, \{category\}, category, f^v_{Books.category}, \{price\}, price, CDNtoUSD)$

$BookSales = deriveAttr(BookSales_v, \{saleAmt\}, salesAmt, CDNtoUSD)$

The derived populations of these two target relations are shown in Figures 4.7 and 4.8. □

MAT-RH supports this step with the Domain Element Environment, DEE. The facilities provided by DEE are the following:

1. It provides templates for the mediator authors to specify the DVFs with appropriate signatures.
2. It allows the mediator author to specify look-up tables.
3. It allows the mediator author to specify the properties of DVFs that are useful for efficient query processing, and their inverses, if they exist.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

| | |
|---|---|
| Travel | Travel and Adventure |
| NewAge | Alternative |
| Computer | Computer and Internet |
| Hobbies | Hobbies |
| Children | Young Readers |

Table 4.1: *categoryMap*: Domain Value Mapping for *Books.category*

| ISBN | title | price | bestSeller | category |
|---|---|---|---|---|
| 001 | "Florida" | 26 | No | "Travel and Adventure" |
| 002 | "China" | 47 | No | "Travel and Adventure" |
| 003 | "Meditation" | 16 | No | "Alternative" |
| 004 | "Dreams" | 11 | Yes | "Alternative" |
| 005 | "TCP/IP" | 23 | Yes | "Computer and Internet" |
| 006 | "HTML" | 20 | Yes | "Computer and Internet" |
| 007 | "Pens" | 42 | No | "Hobbies" |
| 008 | "Quilts" | 21 | No | "Hobbies" |
| 009 | "Micky" | 7 | No | "Young Readers" |
| 010 | "Pooh" | 5 | No | "Young Readers" |

Figure 4.7: Derived population of relation *Books*

| month | salesAmt | book_type |
|---|---|---|
| Feb/99 | 4467 | "hardcover" |
| Mar/99 | 5067 | "hardcover" |
| Feb/99 | 4600 | "paperback" |
| Mar/99 | 5600 | "paperback" |
| Feb/99 | 5333 | "audiobook" |
| Mar/99 | 5200 | "audiobook" |

Figure 4.8: Derived population of relation *BookSales*



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4. It automatically creates the derivation of the target relations, using the DVFs provided by the mediator author.

By now the task of deriving the target view from the source database, as shown in Figure 4.2, is completed using the homogenization methodology and MAT-RH. The contents of the derived relations of *Books* and *BookSales* are shown in Figures 4.7 and 4.8. The next section shows how homogenizing views are combined to derive relations in a service view.

## 4.3 The Integration Framework and AURORA-RI

AURORA's mediation model prescribes that the data sources be first homogenized and then integrated. Integration is the process of deriving data in a pre-defined service view based on data from a set of homogenizing views. The integration framework consists of two components:

1. A registration mechanism. Given a set of homogenizing views, constructed by mediator authors for respective participating sources, integration requires that the mediator author describe the homogenizing view using *registrations* that declare the fragment-of relationship between relations in the homogenizing view and the global relations in the service view. Registration is a means for the mediator author to describe the content of the homogenizing view in the context of a service view.

2. The Match Join operator. Match join is an operator that derives a global relation by combining all the fragments known through registrations.

Integration is performed by AURORA's Relational Integration mediator, AURORA-RI, which accepts registrations from relevant AURORA-RH mediators and realizes the Match Join operator. AURORA-RI is also responsible for entertaining queries against the service view, as described in Chapter 5. The relationship between AURORA-RH and AURORA-RI is illustrated in Figure 4.9.

### 4.3.1 Registrations

A data source contributes data to a service view $S$ by describing the data it offers with a *registration*, sent to the AURORA-RI mediator supporting $S$. A registration is a 3-tuple:

$$REG =< DSN, SR, GRN >$$

where $DSN$ is a data source name, $SR$ is the schema of a relation at $DSN$, $GRN$ is the name of a global relation in $S$. Once the above registration goes through, $SR$ is said to be a *registered fragment* of relation $GRN$. $SR$ must provide the PID of $GRN$ in order to be a fragment of it, that is, $PID(GRN) \subseteq ATTR(SR)$. For any attribute $B$ of $GRN$, $SR$ **supports** $B$ if $B \in ATTR(SR)$. A fragment of a global relation $R$ often supports some, but not all, of the attributes of $R$.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

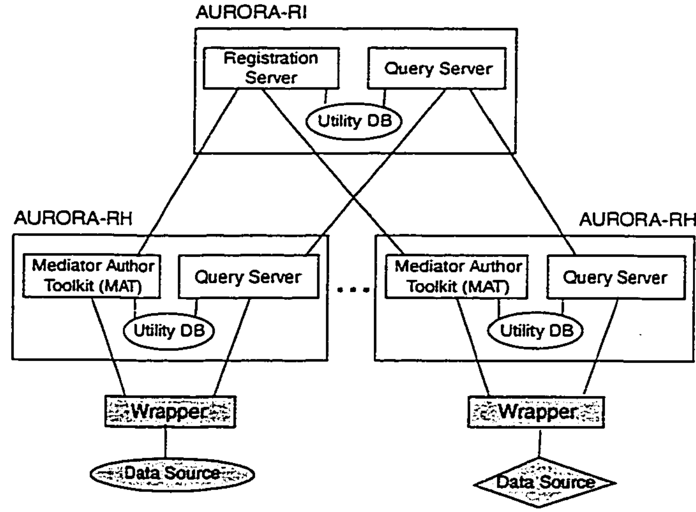



Figure 4.9: Relationship Between Homogenization and Integration Mediators

### 4.3.2 Match Join

For global relation $R_g$ and any of its fragments, $R$, it is mandatory that $PID(R_g) \subseteq ATTR(R)$. Any valid value of $PID(R_g)$, $v$, identifies an application entity, $E_v$. If $v \in R_g\{PID(R_g)\}$ or $v \in R\{PID(R_g)\}$, then $E_v$ is *described* by $R_g$ or $R$, respectively. Assume that $R_g$ is a global relation and $R_1, ..., R_n$ are all the fragments known. Given the content of relations $R_1, ..., R_n$, the derived content of relation $R_g$ must satisfy the following conditions:

1. $\forall v$, $v \in R\{PID(R_g)\}$ if and only if $\exists i, v \in R_i\{PID(R_g)\}$.

2. $\forall t \in R_g, \forall A \in ATTR(R_g), A \notin PID(R_g)$, $\exists a \neq null, t[A] = a$ if and only if $\exists i, \exists t' \in R_i, A \in ATTR(R_i), t'[PID(R_g)] = t[PID(R_g)], t'[A] = a$.

The first condition ensures that any entity described by any fragment is also described by $R_g$, and any entity described by $R_g$ must be described by at least one of its fragments. That is, $R_g$ does not contain "invented" entities. The second condition ensures that, for any entity $E_v$ described by $R_g$ according to the first condition, and a non-PID attribute $A$ of this entity, the value of attribute $A$ of entity $E_v$ as described by $R_g$ should be the same as that described by the fragment(s) that provide a non-null value on $E_v.A$. The value of $E_v.A$ as described by $R_g$ is null if and only if no fragment provides a non-null value for attribute $A$ in its description of entity $E_v$. Notice that if multiple fragments provide distinct, non-null values of $E_v.A$, that is, when there is an instance level conflict over attribute $A$ of entity $E_v$, the second condition above says that all of these values are retained in the global relation, that is, such conflicts are not resolved at integration time. As shown



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

later, these conflicts are resolved at query time, according to application requirements on conflict handling.

In AURORA, global relations are derived using the *Match Join* (MJ) operator, which combines tuples from registered fragments based on PID values. Consider two registered fragments, $F1(P, A, B)$ and $F2(P, B, C)$, of relation $R(P, A, B, C)$ with PID $P$. If $< p, a, b > \in F1$, $< p, b, c > \in F2$, then $< p, a, b, c > \in R$. If $< p, b', c > \in F2$ and $b \neq b'$, then both $< p, a, b, c >$ and $< p, a, b', c >$ are in $R$. MJ can be expressed using outer-joins.

DEFINITION **4.3.1** [Value Set.] Let $Y = \{F_1, ..., F_M\}$ be a set of fragments with a common PID $P$. Let $A_i$ be a non-PID attribute. The *value set of $A_i$ given $Y$*, $VALset(A_i|Y)$, is defined as:

$$VALset(A_i|Y) = \bigcup_{j=1}^{M_i} \pi_{P,A_i}(F_{i_j})$$

where $F_{i_j}$'s ($1 \leq j \leq M_i$) are all the fragments in $Y$ supporting $A_i$. □

$VALset(A_i|Y)$ is a binary relation $(P, A_i)$ containing all the $A_i$-values from the fragments in $Y$ and the related PID value. These binary relations are then outer-joined to derive a global relation.

DEFINITION **4.3.2** [Match Join Operator.] Let $Y = \{F_1, ..., F_M\}$ be a set of fragments with a common PID $P$. Let $S = \{P, A_1, ..., A_g\}$ be a set of attributes, $\forall 1 \leq i \leq g, A_i \neq P$. The *Match Join (MJ)* of relations in $Y$ based on $P$ in regard to $S$ is defined as:

$$MJ(P, S, Y) = VALset(A_1|Y) ⟗_P VALset(A_2|Y) ⟗_P .... ⟗_P VALset(A_g|Y) \quad (4.1)$$

where $⟗_P$ denotes outer-equi-join on $P$. □

DEFINITION **4.3.3** [Global Relation Population.] Let $R$ be a global relation and let $Y_R$ be the set of all fragments registered with $R$, $Y_R = \{F_1, ..., F_M\}$. Then the population of relation $R$ is derived as:

$$R = MJ(PID(R), ATTR(R), Y_R)$$

□

It is easy to verify that the global relations derived according to the above definition satisfy the two conditions given earlier.

### 4.3.3 An Integration Example

**Example 4.3.1** Assume that a service view defines the following global relation, *Books*:

*Books(ISBN, title, year, oprahClub, bestSeller, category, NYTreview, avgReview, price)*

with PID "ISBN". Also assume that *Books* has four registered fragments, as shown in Figure 4.10. According to Definition 4.3.3, AURORA-RI will derive *Books* as shown in Figure 4.11. The column *tid* is not part of the result but is used later to refer to tuples. □



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

| Fragment 1 | | | | |
|---|---|---|---|---|
| ISBN | title | year | oprahClub | |
| 001 | “Florida” | 1960 | No | |
| 002 | “China” | 1966 | No | |
| 003 | “Meditation” | 1972 | Yes | |
| 004 | “Dreams” | 1970 | Yes | |

| Fragment 2 | | | | |
|---|---|---|---|---|
| ISBN | title | oprahClub | bestSeller | |
| 002 | “China” | No | Yes | |
| 003 | “Meditation” | No | Yes | |
| 004 | “Dreams” | Yes | Yes | |
| 005 | “TCP/IP” | Yes | Yes | |

| Fragment 3 | | | | |
|---|---|---|---|---|
| ISBN | title | price | bestSeller | category |
| 001 | “Florida” | 26 | No | “Travel and Adventure” |
| 002 | “China” | 47 | No | “Travel and Adventure” |
| 003 | “Meditation” | 16 | No | “Alternative” |
| 004 | “Dreams” | 11 | Yes | “Alternative” |
| 005 | “TCP/IP” | 23 | Yes | “Computer and Internet” |
| 006 | “HTML” | 20 | Yes | “Computer and Internet” |
| 007 | “Pens” | 42 | No | “Hobbies” |
| 008 | “Quilts” | 21 | No | “Hobbies” |
| 009 | “Micky” | 7 | No | “Young Readers” |
| 010 | “Pooh” | 5 | No | “Young Readers” |

| Fragment 4 | | | | |
|---|---|---|---|---|
| ISBN | title | year | NYTreview | avgReview |
| 001 | “Florida” | 1960 | 1 | 12 |
| 002 | ”China” | 1968 | 5 | 7 |
| 003 | “Meditation” | 1974 | 8 | 7 |
| 004 | “Dreams” | 1980 | 9 | 3 |
| 005 | “TCP/IP” | 1992 | 15 | 15 |
| 006 | “HTML” | 1974 | 16 | 2 |
| 007 | “Pens” | 1986 | 10 | 14 |

Figure 4.10: Registered Fragments for Global Relation Books



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

| tid | ISBN | title | year | oprah-Club | best-Seller | category | NYT-review | avg-Review | price |
|---|---|---|---|---|---|---|---|---|---|
| $t_1$ | 001 | "Florida | 1960 | No | No | "Travel and Adventure" | 1 | 12 | 26 |
| $t_2$ | 002 | "China" | 1966 | No | No | "Travel and Adventure" | 5 | 7 | 47 |
| $t_3$ | 002 | "China" | 1968 | No | No | "Travel and Adventure" | 5 | 7 | 47 |
| $t_4$ | 002 | "China" | 1966 | No | Yes | "Travel and Adventure" | 5 | 7 | 47 |
| $t_5$ | 002 | "China" | 1968 | No | Yes | "Travel and Adventure" | 5 | 7 | 47 |
| $t_6$ | 003 | "Meditation" | 1972 | Yes | No | "Alternative" | 8 | 7 | 16 |
| $t_7$ | 003 | "Meditation" | 1972 | Yes | Yes | "Alternative" | 8 | 7 | 16 |
| $t_8$ | 003 | "Meditation" | 1972 | No | Yes | "Alternative" | 8 | 7 | 16 |
| $t_9$ | 003 | "Meditation" | 1972 | No | No | "Alternative" | 8 | 7 | 16 |
| $t_{10}$ | 003 | "Meditation" | 1974 | Yes | No | "Alternative" | 8 | 7 | 16 |
| $t_{11}$ | 003 | "Meditation" | 1974 | Yes | Yes | "Alternative" | 8 | 7 | 16 |
| $t_{12}$ | 003 | "Meditation" | 1974 | No | Yes | "Alternative" | 8 | 7 | 16 |
| $t_{13}$ | 003 | "Meditation" | 1974 | No | No | "Alternative" | 8 | 7 | 16 |
| $t_{14}$ | 004 | "Dreams" | 1970 | Yes | Yes | "Alternative" | 9 | 3 | 11 |
| $t_{15}$ | 004 | "Dreams" | 1980 | Yes | Yes | "Alternative" | 9 | 3 | 11 |
| $t_{16}$ | 005 | "TCP/TP" | 1992 | Yes | Yes | "Computer and Internet" | 15 | 15 | 23 |
| $t_{17}$ | 006 | "HTML" | 1974 | null | Yes | "Computer and Internet" | 16 | 2 | 20 |
| $t_{18}$ | 007 | "Pens" | 1986 | null | No | "Hobbies" | 10 | 14 | 42 |
| $t_{19}$ | 008 | "Quilts" | 1986 | null | No | "Hobbies" | 10 | 14 | 21 |
| $t_{20}$ | 009 | "Micky" | 1986 | null | No | "Young Readers" | 10 | 14 | 7 |
| $t_{21}$ | 010 | "Pooh" | 1986 | null | No | "Young Readers" | 10 | 14 | 5 |

Figure 4.11: Derived Population of Global Relation *Books*



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

By requiring sources to return fragment data sorted on PID, MJ can be calculated by a multi-way merge-join algorithm. For each PID value $k$, all non-PID attribute values identified by $k$ are collected from all fragments in respective sets, and a Cartesian product of these sets is performed to produce all tuples with PID value $k$. The result of MJ thus computed is sorted on PID.

As discussed earlier, the global relations thus derived may contain instance level conflicts. For example, relation *Books*, as shown in Figure 4.11, gives inconsistent data on the year the book "Meditation" is published. In this sense, AURORA's integration framework produces *conflict-accommodating relations*. These retained conflicts are dealt with at query processing time using a conflict tolerant query model, described in Chapter 5.

## 4.4 Summary

This chapter describes the relational mediation framework of AURORA. It consists of two sub-frameworks: the homogenization framework and the integration framework. Comparison of AURORA with previous work, in terms of the mediation frameworks, can be found in Chapter 2. The homogenization framework consists of a mediation methodology and a set of mediation enabling operators designed for transforming data for homogenization. Compared with previous approaches, this framework not only provides operators that specialize in manipulating heterogeneous data, it also provides a pragmatic means for managing the complexity of working with semantic differences, via the mediator author's toolkit. The relational integration framework has a built-in object-matching facility and retains instance level conflicts, which are exposed to the applications. A conflict tolerant query model is provided to allow the applications to manage these conflicts at query time, as described in the next chapter.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# Chapter 5

# Query Processing in Relational Mediators

From the view point of the applications, a service view is a relational schema that can be queried. regardless of the fact that the data actually originate from multiple heterogeneous data sources. Given a query $Q$ against the service view, the data mediation system is responsible for *decomposing* $Q$ into subqueries against the sources, sending these subqueries to the data sources for execution, collecting the result of these queries, and assembling the answer to $Q$ using these query results.

In AURORA, service views are supported by AURORA-RI mediators that see the sources as a collection of homogenizing views supported by respective AURORA-RH mediators, which in turn access the data sources through the wrappers. A query $Q$ against a service view supported by an AURORA-RI mediator, $M_I$, is processed as follows:

1. $M_I$ decomposes the query into queries against the participating AURORA-RH mediators.
2. Each AURORA-RH mediator that receives subqueries from $M_I$ translates these queries into queries against the underlying data source, submits them for execution through the wrapper, collects the result from the wrapper, assembles the answers to the subqueries submitted by $M_I$, and returns this answer to $M_I$.
3. $M_I$ uses the query results returned by AURORA-RH mediators to assemble an answer to $Q$.

Therefore, AURORA-RI and AURORA-RH cooperate to process queries against the service view, as shown in Figure 4.9. Both AURORA-RI and AURORA-RH perform query decomposition and query answer assembly. While this is the common paradigm of query processing in data mediation systems, AURORA's query processing techniques differ from previous work in the following ways:

1. AURORA-RI supports a conflict tolerant query model, that allows the applications to query



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

potentially inconsistent data. As shown in Section 4.3, the integration framework of AURORA-RI does not resolve instance level conflicts. The reason is that conflict detection and resolution can be costly if done at integration time. In AURORA, conflicts are retained in the integrated relations, and the applications deal with them at query time using the conflict tolerant query model, which provides language constructs to assist the applications in managing the conflicts in large granularities. This query model is described in Section 5.1.

2. AURORA-RH processes queries based on MEA-RH, an algebraic framework specially designed for manipulating data for homogenization. MEA-RH is described in Section 4.2.3. Relations in a homogenizing view are derived using MEA-RH operators. Query processing in AURORA-RH requires query rewriting using view definitions, until a QEP is produced. Intuitively, a QEP is an expression that involves only source relations. The QEP is transformed using transformation rules to produce a more efficient QEP, which is then evaluated. A QEP as an operation tree has leaf nodes that are *retrieve* operators which submit queries to the underlying data source. Non-leaf nodes can be any operator defined by MEA-RH. Query rewriting, transformation rules, and algebraic optimization algorithms are described in Section 5.4.

**Notations.** In this chapter, the following notations are used. $t.A$ is used to denote the value of attribute $A$ in tuple $t$, and $R\{A\}$ to denote all values of attribute $A$ in relation $R$, that is, $R\{A\} = \{a \mid \exists t \in R, t.A = a\}$. Given a collection of relations, $Y = \{F_1, ..., F_m\}$, and an attribute $B$, $Y\{B\} = F_1\{B\} \cup ... \cup F_m\{B\}$. $ATTR(R)$ denotes the set of attributes of relation $R$; $ATTR(p)$ denotes the set of attributes referenced by predicate $p$.

## 5.1 Conflict Tolerant Querying in AURORA-RI

This section describes a technique for querying data in the presence of instance level conflicts. This approach allows applications to control conflict resolution policies at a coarse granularity and gives the system more space for query optimization.

### 5.1.1 Motivation

Traditionally, instance level conflicts are resolved at schema integration time using aggregation functions [21]. Consider a relation *Books* with an attribute *year*, meaning the year a book is published. One may specify that when multiple sources record different *year* for a book, the "correct" *year* value be computed as the average of these values. Queries are written as if data are conflict-free. Conceptually, instance level conflicts are resolved *before* queries are evaluated; users have no say over resolution policies at query time. This approach is referred to as the *static resolution* approach. These resolutions are realized during materialization or query processing. If integrated data are



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

materialized, instance level conflicts are removed before any query is processed. If data are not materialized, that is, they are *virtual*, enough data must be retrieved for conflict detection and resolution at query time; this may incur a significant performance penalty as illustrated by the following example:

**Example 5.1.1** Assume that sources A and B provide data on *Books*, and conflicts on *year* are to be resolved by taking the average of all *year* values. Consider query:

$$Q_0 = \textbf{select } \textit{ISBN, title, category } \textbf{from } \textit{Books } \textbf{where } \textit{year} > 1970$$

It is not sufficient to retrieve only books with *year* > 1970; all *Books* data from both A and B must be retrieved so that the correct *year* values can be computed, and the predicate "*year* > 1970" evaluated. This cost stays the same even when no conflict over *year* actually occurs. Optimization strategies have been proposed [21, 15], but cases such as $Q_0$ are fundamentally difficult to optimize. This drawback becomes significant when more sources contribute large volumes of *Books* data. □

In a dynamic data integration system where large numbers of data sources come and go, materialization may not be desirable. It is also difficult to foresee when and where instance level conflicts are likely to happen; adding a new source may give rise to new conflicts. Specifying a resolution for conflicts that do not really happen incurs unnecessary performance penalties if data are virtual. On the other hand, applications vary in requirements for conflict handling. For $Q_0$ in Example 5.1.1, the exact year of publication of a book does not matter so long as the book is published after 1970. When multiple sources offer different *year* values of a book, one user may consider a book to be published after 1970 if *some* sources say so, while another may require that *all* sources say so. Conflict resolutions on *title* and *category* can be performed only for books that qualified as "published after 1970". Conflicts on *Books.year* is not resolved, but rather *tolerated* by the system during query processing. This approach of instance level conflict handling is referred to as **conflict tolerant querying.**

| Conflicts / Query Evaluation | Statically Resolved | Tolerated |
|---|---|---|
| On Materialized Data | 1 | 3 |
| On Virtual Data | 2 | 4 |

Table 5.1: Querying Integrated Data

Depending on whether integrated data are materialized, and how instance level conflicts are handled, there are 4 cases of querying integrated data, as shown in Table 5.1. Cases 1 and 2 raise no new issues in query semantics; these are well-studied domains. Case 1 requires maintenance of materialized data. Query optimization issues in case 2 have been studied [21, 15]. In AURORA,



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the conflict tolerant query model, the **CT query model**, is defined for use in cases 3 and 4. A framework for reducing redundant data retrieval is developed for use in case 4. Optimizing queries on materialized data in case 3 leverages existing techniques, and is not discussed.

The CT query model enables users to resolve instance level conflicts to a desired degree and let the system "tolerate" the rest; it allows flexible conflict handling and better query performance for users who do not require static resolutions. Consider the following CT query:

| $Q_0' =$ | **select** | *ISBN*, *title*[ANY], *year*[ANY], *category*[DISCARD] |
|---|---|---|
| | **from** | *Books* |
| | **where** | *year* > 1970 **with** HighConfidence |

HighConfidence in the **with** clause specifies that if inconsistent *year* values exist, a book qualifies as *year* > 1970 only if *all* sources say so. *After* a book qualifies, if there is conflict on *title*, *year*, or *category*, the functions ANY, ANY, and DISCARD, respectively, are used to remove these conflicts to produce a conflict-free query answer. Given a set of values $S$, function ANY returns a random value from $S$; function DISCARD returns a null value if $S$ contains more than one distinct value, otherwise it returns the only value in $S$. These resolutions do not affect predicate evaluation; they are used only to produce conflict-free query results. If all sources record that the book "Meditation" is published before 1970, then it does not have to be retrieved even if there is conflict on its *year*. The framework described in Section 5.3 enables such optimized processing.

### 5.1.2 Instance Level Conflicts and Resolutions

In Figure 4.10, Fragment 1 records that the book "Meditation" is published in 1972 while Fragment 4 indicates that the same book is published in 1974. This conflict is reflected in Figure 4.11 as a violation of key constraint, since there is more than one tuple with ISBN 003; these tuples form the *alternative tuple set* for 003.

DEFINITION **5.1.1** [Alternative Tuple Set.] Consider a relation $R$ and a PID value $k$. The *alternative tuple set of $R$ for $k$*, $ATset(R, k)$, is defined as:

$$ATset(R, k) = \{t \mid t \in R, t.PID = k\}$$

□

For example, in *Books* relation given in Figure 4.11, the following can be found:

$ATset(Books, 001) = \{t_1\}$, $ATset(Books, 002) = \{t_2, t_3, t_4, t_5\}$, $ATset(Books, 004) = \{t_{14}, t_{15}\}$

$ATsets$ containing more than one tuple indicate conflicts, as defined below.

DEFINITION **5.1.2** [Conflicts and CA-Relations.] Given a global relation $R$ and a PID value $k$, if $|ATset(R, k)| \geq 2$, then there is a *conflict* in $R$ at $k$. Relations that *may* contain conflicts are called *conflict-accommodating relations*, or CA-relations. □



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Global relations derived according to Definition 4.3.3 are CA-relations, in that they potentially contain conflicts. Formally, conflicts are caused by inconsistencies among registered fragments. Consider two fragments $F_i$, $F_j$ of relation $R$, both supporting a non-PID attribute $A$. If $\exists t_i \in F_i, t_j \in F_j$, $t_i.PID = t_j.PID = k$ but $t_i.A \neq t_j.A$, then by Definition 4.3.2, $\exists t_1, t_2 \in R$, such that $t_1.PID = t_2.PID = k$, $t_1.A \neq t_2.A$. That is, $|ATset(R,k)| \geq 2$. $ATset$ describes conflicts at tuple level. Definition 5.1.3 below describes conflicts at attribute level.

DEFINITION **5.1.3** [Conflicts over attributes.] Given global relation $R$, non-PID attribute $A$, and PID value $k$, there is a conflict on $R.A$ at $k$ if $|ATset(\pi_{PID,A}(R)),k)| \geq 2$. □

Tuple level conflicts result from attribute level ones; resolutions can be performed at both levels. As defined below, a *resolution* in either case is a function with an appropriate signature.

DEFINITION **5.1.4** [Attribute/Tuple Conflict Resolution.] Given a global relation $R$ and its attribute $A$, an *attribute conflict resolution* on $R.A$ is a function $f$:$setof(T) \rightarrow T$, where $T$ is the type of $R.A$. A *tuple conflict resolution* on $R$ is a function $g$ such that, given a set of tuples $S = \{t_1, ..., t_n\} \subseteq R$, $t_i.PID = k$ for $1 \leq i \leq n$, $g(S) = t$ where $ATTR(t) = ATTR(R)$, $t = null$ or $t.PID = k$. □

AURORA provides common functions such as SUM, AVG, MAX, MIN, ANY, DISCARD, but also allows user-defined functions. If conflicts on all attributes are resolved, then effectively a tuple conflict resolution has been performed. This relationship between the two types of resolutions is captured by the concept of *equivalent tuple conflict resolution* (ETCR) given below.

DEFINITION **5.1.5** [Equivalent tuple conflict resolution (ETCR)] Let $R$ be a global relation and $X = \{A_1, ..., A_n\}$ be all the non-PID attributes of $R$ over which there may be conflicts. Let $f_1, ..., f_n$ be attribute conflict resolutions on $A_1, ..., A_n$, respectively. Let $S$ be a set of tuples of $R$ that have the same PID value. A tuple conflict resolution of $R$, $g$, is the *equivalent tuple conflict resolution* of $f_1, ..., f_n$, denoted as $g = ETCR(f_1, ..., f_n)$, if for any set of $R$-tuples with a common PID value, $S$, $g(S) = t$ where $t$ satisfies the following:

1. $\forall i, 1 \leq i \leq n$, $t.A_i = f_i(S_i)$ where $S_i = \{v \mid \exists r \in S, r.A_i = v\}$; and
2. $\forall B \in ATTR(R) - X$, $t.B = r_1.B$, where $r_1 \in S$.

□

Definitions 5.1.6 and 5.1.7 define AURORA's conflict resolution operators: $RAC$ and $RTC$.

DEFINITION **5.1.6** [Operator $RAC$] Let $R$ be a CA-relation and $f_1, ..., f_n$ be conflict resolutions on non-PID attributes $A_1, ..., A_n$. Operator *Resolve Attribute Conflict*, $RAC$, is defined as



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$RAC(R,\ A_1{:}f_1, ..., A_n{:}f_n) = \{t' \mid \ \exists k, t, t \in R,\ t.PID = k,\ t'.PID = k, \\ \forall i, 1 \leq i \leq n,\ t'.A_i = f_i(S(R, A_i, k)), \\ \forall B, B \in ATTR(R) - \{A_1, ..., A_n\},\ t'.B = t.B\}$$

where $S(R, A, k) = \{a \mid\ < k, a > \in \pi_{PID,A}(R)\}$. □

DEFINITION **5.1.7** [Operator *RTC*] Let $R$ be a CA-relation and $F$ be a tuple conflict resolution of $R$. Operator *Resolve Tuple Conflict*, *RTC*, is defined as:

$$RTC(R, F) = \{t \mid \exists k, t = F(ATset(R, k))\}$$

□

Intuitively, $RAC$ removes conflicts on attributes $A_1, ..., A_n$ of $R$ using functions $f_1, ..., f_n$; $RTC$ removes tuple level conflicts using function $F$. These operators are illustrated in Figures 5.1 and 5.2. Given $C$, a set of conflict resolution functions for all the non-PID attributes of $R$ over which there may exist conflicts, $RTC(R, ETCR(C)) = RAC(R, C)$.

| ISBN | title | year | oprah-Club | best-Seller | category | NYT-review | avg-Review | price |
|---|---|---|---|---|---|---|---|---|
| 001 | "Florida | 1960 | No | No | "Travel and Adventure" | 1 | 12 | 26 |
| 002 | "China" | 1966 | No | null | "Travel and Adventure" | 5 | 7 | 47 |
| 003 | "Meditation" | 1972 | Yes | null | "Alternative" | 8 | 7 | 16 |
| 004 | "Dreams" | 1970 | Yes | Yes | "Alternative" | 9 | 3 | 11 |
| 005 | "TCP/TP" | 1992 | Yes | Yes | "Computer and Internet" | 15 | 15 | 23 |
| 006 | "HTML" | 1974 | null | No | "Computer and Internet" | 16 | 2 | 20 |
| 007 | "Pens" | 1986 | null | No | "Hobbies" | 10 | 14 | 42 |
| 008 | "Quilts" | 1986 | null | No | "Hobbies" | 10 | 14 | 21 |
| 009 | "Micky" | 1986 | null | No | "Young Readers" | 10 | 14 | 7 |
| 010 | "Pooh" | 1986 | null | No | "Young Readers" | 10 | 14 | 5 |

Figure 5.1: *RAC* ( *Books*, *year*:MIN, *oprahClub*:ANY, *bestSeller*:DISCARD)

Choice of the strategy for conflict handling is significant. For applications, this strategy impacts on the quality of the data service they receive; it is desirable to have control over this quality. For the mediator, data services of varying qualities incur varying cost; it is desirable to optimize accordingly. In the next section, the Conflict Tolerant (CT) query model - a new approach towards conflict handling - is described. In contrast to previous approaches that either ignore conflicts or require static resolutions, this query model works with CA-relations, but generates conflict-free query results and thus provides **conflict tolerance**.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

| ISBN | title | year | oprah-Club | best-Seller | category | NYT-review | avg-Review | price |
|---|---|---|---|---|---|---|---|---|
| 001 | "Florida | 1960 | No | No | "Travel and Adventure" | 1 | 12 | 26 |
| 002 | "China" | 1966 | No | No | "Travel and Adventure" | 5 | 7 | 47 |
| 003 | "Meditation" | 1972 | Yes | No | "Alternative" | 8 | 7 | 16 |
| 004 | "Dreams" | 1970 | Yes | Yes | "Alternative" | 9 | 3 | 11 |
| 005 | "TCP/TP" | 1992 | Yes | Yes | "Computer and Internet" | 15 | 15 | 23 |
| 006 | "HTML" | 1974 | null | No | "Computer and Internet" | 16 | 2 | 20 |
| 007 | "Pens" | 1986 | null | No | "Hobbies" | 10 | 14 | 42 |
| 008 | "Quilts" | 1986 | null | No | "Hobbies" | 10 | 14 | 21 |
| 009 | "Micky" | 1986 | null | No | "Young Readers" | 10 | 14 | 7 |
| 010 | "Pooh" | 1986 | null | No | "Young Readers" | 10 | 14 | 5 |

Figure 5.2: *RTC(Books*, ANY)

### 5.1.3 Conflict Tolerant Query Model

The semantics of single relation CT queries is defined in this section. A CT query over global relations $R_1, ..., R_n$ is semantically equivalent to a single relation CT query over relation $R_Q = R_1 \times ... \times R_n$. The PID of $R_Q$ includes PIDs of all involved relations. Single relation CT queries are in the following form:

$$Q_{CT} = \begin{array}{ll} \textbf{select} & L \\ \textbf{from} & R \\ \textbf{where} & p \textbf{ with } c_1 \end{array}$$

where $L$ is in one of the following forms:

1. $L = E_1, ..., E_m$ where $E_i = R.B_i$ $(1 \leq i \leq m)$ if $B_i = PID(R)$; $E_i = R.B_i[d_i]$ if $B_i \neq PID(R)$, $d_i$ is an attribute conflict resolution for $R.B_i$.

2. $[D]R.B_1, ..., R.B_m$ where $D$ is a tuple conflict resolution for $\pi_{PID(R),B_1,...,B_n}(R)$.

$c_1$ is called the *predicate evaluation parameter*, or **PE-parameter**, $c_1 \in$ {HighConfidence, RandomEvidence, PossibleAtAll}; it controls how conflicts are handled during predicate evaluation. $d_i$'s and $D$ specify how conflicts are removed to produce a conflict-free query answer. $Q_1$ and $Q_2$ are example CT queries with two forms of select clauses:

$Q_1$:

| | |
|---|---|
| **select** | *ISBN*, *title*[ANY], *year*[ANY], *category*[DISCARD] |
| **from** | *Books* |
| **where** | *year* > 1970 |
| **with** | HighConfidence |

$Q_2$:

| | |
|---|---|
| **select** | [ANY] *ISBN, title, year, category* |
| **from** | *Books* |
| **where** | *year* > 1970 |
| **with** | RandomEvidence |



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Both queries retrieve *ISBN*, *title*, *year* and *category* of books published after 1970. When there is conflict on *year*, $Q_1$ selects books for whom all *year* values available are >1970, while $Q_2$ randomly samples one *year* value and if it is >1970, then the book is selected. After a book qualifies as *year* > 1970, there may still be conflicts on *title*, *year* or *category*; these conflicts are resolved using the resolutions specified in the selection clause. $Q_1$ resolves conflicts on attribute level while $Q_2$ does it on tuple level. The semantics of CT queries is defined in Definitions 5.1.8 and 5.1.10.

A few default forms of $L$ are supported. $L = A_1, ..., A_n$, where $A_i$s are attributes, is the same as $L = [\text{ANY}]A_1, ..., A_n$. If at least one attribute resolution is specified in $L$, the default resolution for all other non-PID attributes with no specified resolution is ANY. Fundamentally, no matter which form $L$ takes, it specifies a tuple conflict resolution, $DE(L)$, referred to as the *data extraction parameter* (the **DE-parameter**). If $L$ is in form 2, $DE(L) = D$. A form 1 select clause can be rewritten into form 2 with $D = ETCR(d_1, ..., d_m)$. From now on, only form 2 select clause is considered.

Definition **5.1.8** [Contributing PID Set.] Given a CA-relation $R$, a predicate $p$ and a PE-parameter $c_1$, the ***contributing PID set*** of $R$ in regard to $p$ under $c_1$, $CSET(R, p, c_1)$, is defined as follows:

1. For any $k \in R\{PID\}$ such that $|ATset(R, k)| = 1$, $k \in CSET(R, p, c_1)$ if and only if $p(t) = true$, where $t \in ATset(R, k)$.
2. For any $k \in R\{PID\}$ such that $|ATset(R, k)| \geq 2$:
   - If $c_1$ = RandomEvidence, $k \in CSET(R, p, c_1)$ if and only if $p(t) = true$, where $t \in ATset(R, k)$ is selected by a function at query evaluation time.
   - If $c_1$ = PossibleAtAll, $k \in CSET(R, p, c_1)$ if and only if $\exists t \in ATset(R, k)$, $p(t) = true$.
   - If $c_1$ = HighConfidence, $k \in CSET(R, p, c_1)$ if and only if $\forall t \in ATset(R, k)$, $p(t) = true$.

□

A $CSET$ contains PIDs identifying tuples that satisfy a predicate under a given PE-parameter; these tuples will contribute to the query result. When the PE-parameter is RandomEvidence, the value of $CSET$ depends on the run-time function used to choose a tuple from an $ATset$, based on which the query predicate is evaluated. Such variations are captured by the following definition.

Definition **5.1.9** [Valid CSET] Let $R$ be a CA-relation, $p$ a predicate, and $c$ a PE-parameter. A set of PID values $C$ is a ***valid value*** for $CSET(R, p, c)$ if:

- $c \neq$ RandomEvidence and $C = CSET(R, p, c)$; or



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- $c$ = RandomEvidence and for any $k \in C$, such that $k \notin CSET(R, p, \text{HighConfidence})$, there exist tuples $t_1, t_2 \in R$, such that $t_1.PID = t_2.PID = k$, $p(t_1) = false$, $p(t_2) = true$.

□

**Example 5.1.2** Examine relation *Books* in Figure 4.11, the following can be found:

$CSET(Books$, "$year$>1973", PossibleAtAll) = {003, 004, 005, 006, 007, 008, 090, 010}
$CSET(Books$, "$year$>1973", HighConfidence) = {005, 006, 007, 008, 090, 010}
$CSET(Books$, "$year$>1973", RandomEvidence) = {004, 005, 006, 007, 008, 090, 010}
$CSET(Books$, "$year$>1973", RandomEvidence) = {003, 004, 005, 006, 007, 008, 090, 010}

The last two $CSET$s given above are both valid. 003 does not satisfy $year > 1973$ under HighConfidence because there is evidence in relation *Books* that the book 003, "Meditation", is published in 1972. □

DEFINITION **5.1.10** [Answer Set.] The answer to query $Q_{CT}$ given earlier is defined as:

$$A = \pi_{B_1,\ldots,B_m}[RTC(R \ltimes_{PID} CSET(R, p, c_1),\ DE(L))]$$

□

Tables 5.2 and 5.3 show 12 CT queries and results. These queries vary in PE-parameter and DE-parameter. Two DE-parameters are illustrated: ANY and DISCARD. The attributes in the select clause are also varied to demonstrate how the CT query model tolerates conflicts. Results of queries involving RandomEvidence or ANY may vary with the selection function used at run-time. By specifying these parameters, one accepts such variations.

**Example 5.1.3** First examine $Q_1$-$Q_6$ and their results shown in Table 5.2, and observe how various conflict handling policies impact on the query results. The most rigorous control appears in $Q_2$. This query has the smallest $CSET$ and one of the smallest results. Next, observe that queries in Table 5.3 often have larger results. For example, $Q_4'$ has a larger result than $Q_4$ because relation *Books* contains no conflicts over *title* but it contains conflicts over *year*. This shows that when a query retrieves only conflict-free attributes, conflicts on other attributes are well tolerated by the system and are often hidden from the users. □

## 5.2 Primitive Evaluation of Conflict Tolerant Queries

Algorithm *CT-QP-NoOpt* is an unoptimized algorithm that directly implements the CT query semantics given in Definitions 5.1.8 and 5.1.10. Correctness of this algorithm is straightforward.

**ALGORITHM** *CT-QP-NoOpt* ($R$, $Q_{CT}$, $F_1, \ldots, F_n$)
**input:**

$R$: Global relation involved in the query.
$Q_{CT}$: $Q_{CT}$ = **select** $L$ **from** $R$ **where** $p$ **with** $c_1$.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

| Query | Answer |
| --- | --- |
| $Q_1$ :<br>**select** [ANY] *title*, *year*<br>**from** *Books*<br>**where** 1967 < *year* < 1985<br>**with** HighConfidence | < "Meditation", 1972 ><br>< "Dreams", 1980 ><br>< "HTML", 1984 > |
| $Q_2$ :<br>**select** [DISCARD] *title*, *year*<br>**from** *Books*<br>**where** 1967 < *year* < 1985<br>**with** HighConfidence | < "HTML", 1974 > |
| $Q_3$ :<br>**select** [ANY] *title*, *year*<br>**from** *Books*<br>**where** 1967 < *year* < 1985<br>**with** PossibleAtAll | < "China", 1968 ><br>< "Meditation", 1972 ><br>< "Dreams", 1980 ><br>< "HTML", 1974 > |
| $Q_4$ :<br>**select** [DISCARD] *title*, *year*<br>**from** *Books*<br>**where** 1967 < *year* < 1985<br>**with** PossibleAtAll | < "HTML", 1974 > |
| $Q_5$ :<br>**select** [ANY] *title*, *year*<br>**from** *Books*<br>**where** 1967 < *year* < 1985<br>**with** RandomEvidence | < "Meditation", 1974 ><br>< "Dreams", 1980 ><br>< "HTML", 1974 > |
| $Q_6$ :<br>**select** [DISCARD] *title*, *year*<br>**from** *Books*<br>**where** 1967 < *year* < 1985<br>**with** RandomEvidence | < "HTML", 1974 > |

Table 5.2: Example Queries and Answers

$F_i$'s: All the fragments registered with $R$, $F_1, ..., F_n$.
**output:** $A$: the query answer.
**begin**

1. $R = \pi_{L_1} MJ(PID(R), ATTR(R), F_1, ..., F_n)$, where $L_1 = \{PID(R)\} \cup ATTR(p) \cup ATTR(L)$.
2. $C = ComputeCSET(R, p, c_1)$, where $c_1$ is the PE-parameter of $Q_{CT}$.
3. $A = \pi_{ATTR(L)}[RTC(\pi_{ATTR(L)\cup\{PID\}}(R \ltimes_{PID} C), DE(L))]$;

**end of algorithm.**

In step 1, *CT-QP-NoOpt* retrieves all fragments and performs a match join. This can be expensive when fragments are large and numerous. When query selectivity is low, a large portion of the retrieved data is discarded in step 2, where $CSET$ is computed with the algorithm given below; it is desirable to not retrieve these data in step 1. In step 3, operator $RTC$ is applied to produce a conflict-free result. $RTC$ is a direct implementation of Definition 5.1.7 and is not given here.

**ALGORITHM** *ComputeCSET*$(R, p, c_1)$
**input:**
$R$: A CA-relation, sorted on $PID(R)$.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

| Query | Answer |
|---|---|
| $Q'_1$ :<br>**select** [ANY] *title*<br>**from** *Books*<br>**where** 1967 $<$ *year* $<$ 1985<br>**with** HighConfidence | < “Meditation” ><br>< “Dreams” ><br>< “HTML” > |
| $Q'_2$ :<br>**select** [DISCARD] *title*<br>**from** *Books*<br>**where** 1967 $<$ *year* $<$ 1985<br>**with** HighConfidence | < “Meditation” ><br>< “Dreams” ><br>< “HTML” > |
| $Q'_3$ :<br>**select** [ANY] *title*<br>**from** *Books*<br>**where** 1967 $<$ *year* $<$ 1985<br>**with** PossibleAtAll | < “China” ><br>< “Meditation” ><br>< “Dreams” ><br>< “HTML” > |
| $Q'_4$ :<br>**select** [DISCARD] *title*<br>**from** *Books*<br>**where** 1967 $<$ *year* $<$ 1985<br>**with** PossibleAtAll | < “China” ><br>< “Meditation” ><br>< “Dreams” ><br>< “HTML” > |
| $Q'_5$ :<br>**select** [DISCARD] *title*<br>**from** *Books*<br>**where** 1967 $<$ *year* $<$ 1985<br>**with** RandomEvidence | < “Meditation” ><br>< “Dreams” ><br>< “HTML” > |
| $Q'_6$ :<br>**select** [DISCARD] *title*<br>**from** *Books*<br>**where** 1967 $<$ *year* $<$ 1985<br>**with** RandomEvidence | < “China” ><br>< “Meditation” ><br>< “Dreams” ><br>< “HTML” > |

Table 5.3: Example Queries and Answers

$p$: A predicate.

$c_1$: A PE parameter.

**output:** $C$: $CSET(R, p, c_1)$.
**begin**

1. Let $R' = \sigma_p(R)$.
2. If $c_1$ = RandomEvidence or $c_1$ = PossibleAtAll, then $C = \pi_{PID}(R')$.
3. If $c_1$ = HighConfidence, then $C = \pi_{PID}(R') - \pi_{PID}(R - R')$.

**end of algorithm.**

In the next section, techniques are established to use query predicate $p$ to derive conditions based on which enough data are retrieved from fragments to guarantee correct query evaluation, but data that do not contribute to the query result are not retrieved. This technique reduces both communication cost and the volume of data manipulated at the mediator.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

## 5.3 Optimized Processing of Conflict Tolerant Queries

For any predicate $p$ over a global relation $R$ and a given fragment of $R$, $F$, if $ATTR(p) \subseteq ATTR(F)$, then $p$ is **applicable** to $F$. The goal of CT query optimization is to use applicable predicates to reduce the volume of fragment data retrieved while preserving query semantics.

### 5.3.1 CT Query Optimization: an Overview

Let $p$ be a predicate over $R$ and let $p = p_1 \wedge ... \wedge p_m$ be its conjunctive normal form. Given a registered fragment of $R$, $F$, the question is: *if* $p_x (1 \leq x \leq m)$ *is applicable to* $F$, *can one retrieve only* $\sigma_{p_x} F$ *into the mediator and still evaluate* $CSET(R, p, c)$ *correctly?*

Consider the fragments shown in Figure 4.10 and $C = CSET(Books, \text{"}year > 1973\text{"}, c)$. Assume that only the following is retrieved into the mediator: $\sigma_{year>1973}$ (Fragment 1) and $\sigma_{year>1973}$ (Fragment 4). $t_{meditation}$ = (003, "Meditation", 1972, Yes) in Fragment 1 will not be retrieved. This potentially excludes 003 from $C$. If $c$ = RandomEvidence, it is valid to exclude 003 from $C$, according to Definition 5.1.9. If $c$ = HighConfidence and if 003 is excluded from $C$ then it is correct. However, the mediator will retrieve $t_{meditation2}$ = (003, "Meditation", 1974, 8, 7) from Fragment 4, and algorithm *ComputeCSET* would include 003 in $C$. This is incorrect. To solve this problem, one can send 003 to the site of Fragment 1 to verify that the book "Meditation" indeed has $year > 1973$. In our example, the verification fails and 003 is removed from $C$. This process is referred to as *PID verification.* Obviously, when *year* is supported by only one fragment, PID verification is not needed. Assume that *ComputeCSET* derives a temporary $CSET$ value $C'$ from reduced fragments, PID verification can be performed by sending the following queries to the sites of Fragment 1 and 4, respectively:

$$\delta_1 = C' \cap \pi_{ISBN} \sigma_{year \leq 1973}(\text{Fragment 1}); \quad \delta_4 = C' \cap \pi_{ISBN} \sigma_{year \leq 1973}(\text{Fragment 4})$$

PID values in $\delta_1$ or $\delta_4$ must be removed from $C'$. The cost of this approach is low when (1) query selectivity is low resulting in a small $C'$; and (2) Conflict rate is low resulting in small $\delta$s. When no conflict exists, all $\delta$s are empty. When $C'$ is large, the cost of PID verification may defeat the savings achieved by pushing selections onto fragments; a cost model is needed for strategy selection.

If $c$ = PossibleAtAll, $C$ can be computed by *ComputeCSET* correctly from reduced fragments. However, pushing predicates that involve more than one attribute is not as straightforward. Consider $C_1$ = *CSET(Books, "oprahClub = bestSeller",* PossibleAtAll*)*. In Figure 4.10, Fragment 2 contains tuple (003, "Meditation", No, Yes). If only $\sigma_{oprahClub=bestSeller}$ (Fragment 2) is retrieved, 003 will be excluded from $C_1$. This is incorrect, since combining Fragments 1 and 2, it is possible that the book "Meditation" is both an Oprah's club book and a best seller. Generally, pushing a multi-attribute predicate $p$ onto a fragment $F$ is possible only if no fragments other than $F$ support any of the attributes involved in $p$.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CT query optimization possibilities are summarized in Table 5.4. In the rest of this section, the above described optimization strategies are formally established. When $c$ = HighConfidence, a cost model is needed to determine whether the strategies devised here actually reduce cost. This is a future research issue; for now the validity of the strategies is established.

| | Can $p_x$ be used for fragment reduction? |
|---|---|
| $c$ = RandomEvidence | YES |
| $c$ = PossibleAtAll | YES (Conditional) |
| $c$ = HighConfidence | YES (with PID verification) |

Table 5.4: Fragment Reduction with Selections

### 5.3.2 A Theory of Conflict Tolerant Query Processing

The main theorems of our theory are Theorems 5.3.1 and 5.3.2, which allow us to push selections across MJ onto fragments, to various degrees, according to the PE-parameter.

THEOREM 5.3.1 *Let $R$ be a CA-relation. Let $p = p_1 \wedge p_2 \wedge ... \wedge p_x$ be a predicate over $R$ in conjunctive normal form. Let $F_1, ..., F_n$ be all fragments registered with $R$ that contain no null values. Let $p^i = p_1^i \wedge ... \wedge p_{s_i}^i$, where $p_j^i \in \{p_1, ..., p_x\}, 1 \leq j \leq s_i$ is applicable to $F_i$. Let $F_i' = \sigma_{p^i}(F_i)$, $1 \leq i \leq n$. Let $T_i = \sigma_{\neg p^i}(F_i)$, $1 \leq i \leq n$. Let $W = T_1\{PID\} \cup ... \cup T_n\{PID\}$. Let $R' = MJ(PID(R), ATTR(R), F_1', ..., F_n')$. Then the following is true:*

1. *CSET($R'$, p, RandomEvidence) is a valid value for CSET(R, p, RandomEvidence);*
2. *CSET(R, p, HighConfidence) = CSET($R'$, p, HighConfidence) − W.*

Note that point 2 of Theorem 5.3.1 says that $CSET(R, p, \text{HighConfidence})$ can be computed from reduced fragments, but one must verify that the PID values thus selected are not in any $T_x$. This process is called **PID verification.**

THEOREM 5.3.2 *Let $R$ be a CA-relation. Let $p = p_1 \wedge p_2 \wedge ... \wedge p_x$ be a predicate over $R$ in conjunctive normal form. Let $F_1, ..., F_n$ be all fragments registered with $R$. $F_i$s do not contain null values. Then*

$$CSET(R, p, PossibleAtAll) = CSET(MJ(R, F_1', ..., F_n'), p, PossibleAtAll)$$

*where $\forall i, 1 \leq i \leq n$, $F_i' = \sigma_{p^i}(F_i)$, $p^i = p_1^i \wedge ... \wedge p_{s_i}^i$, $p_j^i \in \{p_1, ..., p_x\}, 1 \leq j \leq s_i$; $p^i$ satisfies the following:*

1. *$ATTR(p^i) = \{PID\}$ or $ATTR(p^i) = ATTR(p) \cap ATTR(F_i)$; and*
2. *$p_j^i (1 \leq j \leq s_i)$ involves at most one non-PID attribute, or no registered fragment of $R$ other than $F_i$ supports any of the non-PID attributes in $ATTR(p_j^i)$.*



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In the rest of this section, the proofs of the above theorems are presented. First, 5 theorems on basic properties of the MJ operator are given. Proof of these theorems are mostly by definition.

THEOREM 5.3.3 *Let $R$ be a global relation and $F_1, ..., F_n$ be all the fragments registered with $R$. Let $L$ be a list of attributes such that $PID(R) \in L$. Then:*

$$\pi_L(R) = MJ(PID(R), L, \pi_{L_1}(F_1), ..., \pi_{L_n}(F_n))$$

*where $L_i = L \cap ATTR(F_i), 1 \leq i \leq n$.*

This theorem allows us to push projections onto fragments. The projection list must include the PID.

THEOREM 5.3.4 *Let $R = MJ(P, S, Y)$ where $Y = \{F_1, ..., F_n\}$, $S = \{A_1, ..., A_m\}$, and $P$ is the common PID of all fragments in $Y$. Then:*

$$R\{P\} = F_1\{P\} \cup ... \cup F_n\{P\} = VALset(A_1|Y)\{P\} \cup ... \cup VALset(A_m|Y)\{P\}$$

This theorem describes the relationship among the PID values in a CA-relation, those in the fragments, and those in the value sets; this relationship is used to prove later theorems.

THEOREM 5.3.5 *Let $R = MJ(P, S, Y)$ and let $S_1 \subseteq S$ be a set of attributes, $S_1 = \{A_1, ..., A_d\}$, and $P$ is the common PID of all fragments in $Y$. Let $a_1, ..., a_d$ be a set of non-null values and let $k$ be a PID value. Then the following two statements are equivalent:*

1. *$\exists t \in R$ such that $t.P = k$, $t.A_j = a_j, 1 \leq j \leq d$.*
2. *$\forall j, 1 \leq j \leq d$, if $A_j \neq P$, then $< k, a_j > \in VALset(A_j|Y)$.*

This theorem describes the relationship between the content of the value sets and tuples in the global relation computed using these value sets; this relationship is used to prove later theorems.

THEOREM 5.3.6 *Let $R = MJ(P, S, F_1, ..., F_n)$, where $P$ is the common PID of fragments $F_i(1 \leq i \leq n)$. Let $R' = MJ(P, S, F_1 - T_1, ..., F_n - T_n)$ where $T_i \subseteq F_i$. Let $W = T_1\{P\} \cup ... \cup T_n\{P\}$. For any PID value $k' \in (R\{P\} - W)$, $k' \in R'\{P\}$ and $ATset(R, k') = ATset(R', k')$.*

This theorem says that if one does not retrieve a portion of each fragment, the $T_i$s, before performing MJ, the data related to PID values in $R\{P\} - W$ are not affected.

THEOREM 5.3.7 *Let $R = MJ(P, S, F_1, ..., F_n)$, where $P$ is the common PID of fragments $F_i(1 \leq i \leq n)$, and let $T_i \subseteq F_i(1 \leq i \leq n)$. Given a predicate $p$ and a tuple $t' \in MJ(P, S, F_1 - T_1, ..., F_n - T_n)$, if $\forall A \in ATTR(p), t'.A \neq null$, then $\exists t \in R$ such that $t.P = t'.P$ and $p(t) = p(t')$.*



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This describes a condition that is weaker than the following:

$$MJ(P, S, F_1 - T_1, ..., F_n - T_n) \subseteq MJ(P, S, F_1, ..., F_n)$$

which is in fact not always true.

DEFINITION **5.3.1** [Images.] Let $R$ be a CA-relation with PID, $P$, and $F$ be a registered fragment of $R$. Given attribute set $S \subseteq ATTR(F)$, and tuple $t_f \in F$, $t \in R$ is an *image of* $t_f$ *over* $S$, denoted as $t \in imagesOf(t_f, S, F, R)$, if:

1. $S = \{P\}$ and $t.P = t_f.P$; or

2. $t.P = t_f.P$ and there exists a non-PID attribute $A \in S$, such that $t.A = t_f.A$.

□

Images of a given source tuple $t_f$ are all the tuples in the global relation that $t_f$ *may* contribute to. For example, let $t_f$ be tuple 004 in Fragment 1 in Figure 4.10, then: $imagesOf(t_f$,{"ISBN"},"Fragment 1",Books) = $\{t_{14}, t_{15}\}$, $imagesOf(t_f$,{"year"},"Fragment 1",Books) = $\{t_{14}\}$

DEFINITION **5.3.2** [Irrelevant Source Tuples.] Let $R$ be a CA-relation and $F$ be a fragment registered with $R$, $p$ be a predicate. $t_r \in F$ is an *irrelevant source tuple of* $R$ in regard to $p$ if $\exists t \in R$, such that $t.PID = t_r.PID$ and $p(t) = false$. □

DEFINITION **5.3.3** [Negative Source Tuples.] Let $R$ be a CA-relation and $F$ be a fragment registered with $R$, $p$ be a predicate. $t_v \in F$ is a *negative source tuple of* $R$ in regard to $p$ if $\forall t \in imagesOf(t_v, ATTR(p), F, R)$, $p(t) = false$. □

Intuitively, not retrieving the irrelevant tuples does not impact on the correctness of CT query processing. Negative source tuples are those that definitely will not contribute to the final query result and hence should never be retrieved.

THEOREM 5.3.8 *Let* $R$ *be a CA-relation and* $F_1, ..., F_n$ *be fragments registered with* $R$. *Let* $T_1, ..., T_n$ *be sets of irrelevant source tuples in regard to* $p$, $T_i \subseteq F_i$. *Let* $R' = MJ(PID(R), ATTR(R), F_1 - T_1, ..., F_n - T_n)$. *Then:*

1. $CSET(R', p, RandomEvidence)$ *is a valid value for* $CSET(R, p, RandomEvidence)$.

2. *CSET(R,p,HighConfidence) = CSET( R', p, HighConfidence) - W, where* $W = \{k | \exists i, k \in T_i\{PID(R)\}\}$.

**Proof:** Consider $C$, a valid value for $CSET(R', p,$ RandomEvidence) and $k \in C$, $k \notin$ *CSET(R, p,* HighConfidence*)*. Since $k \in C$, $\exists$ $t \in$ *MJ(PID(R), ATTR(R),* $F_1 - T_1, ..., F_n - T_n$*)*, such that $t.PID = k$, $p(t) = true$. By Theorem 5.3.7, $\exists t_1 \in R$, such that $t_1.PID = k$ and $p(t_1) = true$. Since



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$k \notin$ *CSET(R, p,* HighConfidence*)*, $\exists t_2 \in R$ such that $t_2.PID = k$ and $p(t_2) = false$. By Definition 5.1.9, $C$ is a valid value for *CSET(R, p, c)*.

Consider $k \in$ *CSET(R, p,* HighConfidence*)* $- W$. By Theorem 5.3.6, since $k \notin W$, $ATset(R.k) = ATset(R',k)$. By definition, $k \in$ *CSET(R', p,* HighConfidence*)* $- W$. Thus *CSET(R, p,* HighConfidence*)* $- W \subseteq$ *CSET(R', p,* HighConfidence*)* $- W$. However, $\forall k \in W$, $\exists t_i \in T_i$ such that $t_i.PID = k$ and $t_i$ is an irrelevant source tuple in regard to $p$. By definition, $\exists t \in R$, $t.PID = k$, $p(t) = false$. Thus $k \notin$ *CSET(R, p,* HighConfidence*)*. Thus *CSET(R, p,* HighConfidence*)* $- W =$ *CSET(R, p,* HighConfidence*)*. Thus,

*CSET(R, p,* HighConfidence*)* $\subseteq$ *CSET(R', p,* HighConfidence*)* $- W$

Now consider $k \in$ *CSET( R', p,* HighConfidence*)* $- W$. Since $k \notin W$, from Theorem 5.3.6, $ATset(R',k) = ATset(R,k)$. Hence $k \in$ *CSET(R, p,* HighConfidence*)*. Thus,

*CSET( R', p,* HighConfidence*)* $- W \subseteq$ *CSET(R, p,* HighConfidence*)*

Theorem is proven. ∎

THEOREM 5.3.9 *Let $R$ be a CA-relation and $F_1, ..., F_n$ be fragments registered with $R$. Let $T_1, ..., T_n$ be sets of negative source tuples in regard to predicate $p$, $T_i \subseteq F_i$. Let*

$$R' = MJ(PID(R), ATTR(R), F_1 - T_1, ..., F_n - T_n)$$

*Then,*

$$CSET(R, p, PossibleAtAll) = CSET(R', p, PossibleAtAll)$$

.

**Proof:** Denote $Y = \{F_1, .., F_n\}$, $Y_T = \{T_1, ..., T_n\}$, $Y' = \{F_1 - T_1, ..., F_n - T_n\}$. Obviously, $Y\{PID\} - Y_T\{PID\} \subseteq Y'\{PID\}$. Consider $k \in$ *CSET(R, p,* PossibleAtAll*)* and show that $k \in$ *CSET(R', p,* PossibleAtAll*)*. By definition, $\exists\ t \in R$, $t.PID = k$, $p(t) = true$. Let $ATTR(p) = \{A_1, ..., A_m\}$ and let $a_j = t.A_j, 1 \le j \le m$, $a_j \ne null$. If $A_j = PID(R)$, then $a_j = k$. First show that $k \in Y'\{PID\}$ and that $\forall j, 1 \le j \le m$, $A_j = PID(R)$ or $<k, a_j> \in VALset(A_j|Y')$. Consider two cases:

**case 1.** $ATTR(p) = \{PID(R)\}$, by theorem assumption, $\forall t' \in R$ such that $\exists i, t'.PID \in T_i(PID(R))$, $p(t') = false$. Hence $k \in Y\{PID\} - Y_T\{PID\}$. Since $Y\{PID\} - Y_T\{PID\} \subseteq Y'\{PID\}$, $k \in Y'\{PID\}$.

**case 2.** $ATTR(p)$ includes a non-PID attribute $A_x (1 \le x \le m)$. Consider $<k, a_x>$. Since $p(t) = true$, by theorem assumption, $t$ is not an image of any tuple in any $T_i$ over $ATTR(p)$. Hence

$$<k, a_x> \in VALset(A_x|Y) - VALset(A_x|Y_T)$$

Since $VALset(A_x|Y) - VALset(A_x|Y_T) \subseteq VALset(A_x|Y')$, the following holds:

$$<k, a_x> \in VALset(A_x|Y')$$



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Hence, $k \in Y'\{PID\}$. By Theorem 5.3.5, $\exists t' \in R'$ such that $\forall j, 1 \leq j \leq m$, $t'.A_j = a_j = t.A_j, t'.A_j \neq null$. Obviously, $p(t') = p(t) = true$. Hence, $k \in CSET(R', p,$ PossibleAtAll$)$. So it is shown that $CSET(R, p,$ PossibleAtAll$) \subseteq CSET(R', p,$ PossibleAtAll$)$. Now consider $k \in CSET(R', p,$ PossibleAtAll$)$. $\exists t \in R'$ such that $t.PID = k$, $p(t) = true$. By Theorem 5.3.7, $\exists t' \in R$, such that $t'.PID = k$, $p(t') = p(t) = true$, hence $k \in CSET(R, p,$ PossibleAtAll$)$. So it is shown that $CSET(R', p, PossibleAtAll) \subseteq CSET(R, p, PossibleAtAll)$. Theorem proven.

∎

**Proof of Theorem 5.3.1**

**Proof:** For any $i, 1 \leq i \leq n$, since $F_i$ contains no null values, $F_i' = F_i - T_i$. Now show that all tuples in $T_i$ are irrelevant source tuples in regard to p by showing that $\exists t_R \in R$ such that $t_R.PID = t.PID$, $p(t_R) = false$. The rest of the theorem follows from Theorem 5.3.8.

Consider $t_i \in T_i$, $p^i(t) = false$. Let $ATTR(p^i) = \{B_1, ..., B_x\}$, $k = t_i.PID$ and $b_j = t_i.B_j$, $b_j \neq null, 1 \leq j \leq x$. Since $T_i \subseteq F_i$, $k \in R\{PID\}$. By Theorem 5.3.5, $\exists t_R \in R$ $t_R.PID = k$, $t_R.B_i = t_i.B_i, 1 \leq i \leq x$. Obviously, $p^i(t_R) = p^i(t_i) = false$. Thus, $p(t_R) = false$, and $t_i$ is an irrelevant source tuple in regard to p. Theorem proven.

∎

**Proof of Theorem 5.3.2**

**Proof:** Let $T_i = \sigma_{\neg p^i}(F_i)$, $i, 1 \leq i \leq n$, since $F_i$ contains no null values, $F_i' = F_i - T_i$. Now show $T_i$ is a set of negative source tuples in regard to p by showing that $\forall t_R \in R, \forall t_i \in T_i$, such that $t_R \in imagesOf(t_i, ATTR(p), F_i, R)$, $p(t_R) = false$. The rest of the theorem follows from Theorem 5.3.9. Let $t_i \in T_i$ and consider the following cases:

**case 1.** $ATTR(p^i) = \{PID\}$. If $t_R$ is an image of $t_i$ *in* $R$, $t_R.PID = t_i.PID$. $p^i(t_R) = p^i(t_i) = false$.

**case 2.** $ATTR(p^i)$ involves non-PID attributes, but $ATTR(p^i) = ATTR(p) \cap ATTR(F_i)$. Consider an image of $t_i$, $t_R \in R$ over $ATTR(p)$. By Definition 5.3.1, $t_R.PID = t_i.PID$, and $\exists B \in ATTR(p) \cap ATTR(F_i)$, such that $t_R.B = t_i.B$. Since $ATTR(p^i) = ATTR(p) \cap ATTR(F_i)$, $B \in ATTR(p^i)$, that is, $\exists j, 1 \leq j \leq s_i$, $B \in ATTR(p^i_j)$. Consider two cases:

1. $p^i_j$ involves only one non-PID attribute, $B$. $p^i_j(t_R) = p^i_j(t_i) = false$. Hence $p(t_R) = false$.

2. $p^i_j$ involves more than one non-PID attribute $B, B_1, ..., B_y$, but no fragments other than $F_i$ support any of them. $\forall i, 1 \leq i \leq y, t_R.B_i = t_i.B_i$. Thus, $p^i_j(t_R) = p^i_j(t_i) = false$. Thus, $p(t_R) = false$.

Theorem proven. ∎



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

### 5.3.3 Optimized Conflict Tolerant Query Evaluation Algorithms

The following algorithm is directly based on Theorems 5.3.1 and 5.3.2.

**ALGORITHM** *Optimized-CT-QP* ($R$, $Q_{CT}$, $F_1, ..., F_n$)
**input:**

$R$: Global relation $R$ involved in the query.

$Q_{CT}$: $Q_{CT}$ = **select** $L$ **from** $R$ **where** $p$ **with** $c_1$.

$F_i$'s: All the fragments registered with $R$.

**output:** $A$: the query answer.

**begin**

**Compute CSET:**

- Let $L_1 = ATTR(L) \cup \{PID(R)\} \cup ATTR(p)$. Write $p$ into conjunctive normal form $p = p_1 \cap ... \cap p_x$. Let $X_p = \{p_1, ..., p_x\}$.
- For $i = 1, n$ do:
  - if $c_1 \neq$ *PossibleAtAll* then let $p^i$ be the conjunction of all predicates in $X_p$ that are applicable to $F_i$. If no such $p^i$ is found, $p^i = true$;
  - if $c_1$ = PossibleAtAll then let $p^i$ be the conjunction of all predicates in $X_p$ such that (1) it involves at most one non-PID attribute; or (2) No fragments other than $F_i$ supports any of the non-PID attributes involved. If $ATTR(p^i) \neq ATTR(p) \cap ATTR(F_i)$, $p^i = true$.

  **S1** $F_i' = \pi_{L_1 \cap ATTR(F_i)} \sigma_{p^i}(F_i)$.
- $R' = MJ(PID(R), L_1, F_1', ..., F_n')$;
- $C = ComputeCSET(R', p, c_1)$;

**PID Verification:**

- If $c_1$ = HighConfidence or $DE(L) \neq ANY$ then
  - Let $L_2 = ATTR(L) \cup \{PID(R)\}$.
  - For $i = 1, n$ do:

    **S2** Let $\delta_i = \pi_{L_2 \cap ATTR(F_i)} \sigma_{\neg p^i}(F_i \bowtie_{PID(R)} C)$;
    - if $c_1$ = HighConfidence
      $C = C - \delta_i(PID(R))$; $\delta_i = \emptyset$;
- $R' = R' \bowtie_{PID(R)} C$;

**Data Completion:**

- if DE(L) $\neq$ ANY then $R'$ = MJ(PID(R), $L_2$, $R'$, $\delta_1, ..., \delta_n$);

**Data Extraction:**

- $A = \pi_{ATTR(L)}[RTC(R' \bowtie_{PID(R)} C, DE(L))]$.

**end of algorithm.**

Steps S1 and S2 are where queries are sent to the data sources that provide the respective fragments. These steps follow directly from Theorems 5.3.1 and 5.3.2. When the number of sources involved is large and data volume is large, cutting down on data retrieval at S1 and S2 improves query performance. Moreover, the following observations can be made:

**Optimized-CT-QP is a 1- or 2-phase algorithm.** The first phase retrieves enough data to compute $CSET$. Depending on the PE- and the DE-parameter, a second phase retrieves extra data



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

for PID verification and/or data completion. PID verification is needed only if the PE-parameter is HighConfidence. Data completion is not needed when the DE-parameter is ANY.

**Performance perspectives of Optimized-CT-QP.** Step S1 is obviously a good move towards saving communication cost. At step S2, one could send the content of the computed $CSET$, $C$, to relevant data sources. This works well when $C$ is small due to a low query selectivity, but may get expensive when $C$ is large. A simple computation can be applied to restrict this cost. Consider performing step S2 against a data source supporting a fragment $F_i$. The purpose of sending a query to compute $W_i$ is to retrieve data related to PIDs in $F_i$ that are in $C$, but have not been retrieved in step S1. Thus, one can compare the volume of $\pi_{L_2 \cap ATTR(F_i)}\sigma_{\neg p^i}(F_i)$ with the volume of $C$. If the former is smaller, it is simply retrieved without sending $C$ to the relevant data source, and the process continues normally.

**Optimized-CT-QP performs better when conflict rate is low.** When conflict rate is low, the $\delta_i$'s will be empty or very small. This means the cost of PID verification and Data Completion becomes low. Therefore, *Optimize-CT-QP* is expected to be most efficient against low conflict data.

| $F_1' = \pi_{ISBN,title,year,oprahClub}\sigma_{year>1973}(F_1)$ | | | | $F_2' = \pi_{ISBN,title,oprahClub,bestSeller}\sigma_{oprahClub=bestSeller}(F_2)$ | | | |
|---|---|---|---|---|---|---|---|
| ISBN | title | year | oprahClub | ISBN | title | oprahClub | bestSeller |
| | | | | 004 | "Dreams" | Yes | Yes |
| | | | | 005 | "TCP/IP" | Yes | Yes |

| $F_3' = \pi_{ISBN,title,bestSeller}(F_3)$ | | | $F_4' = \pi_{ISBN,title,year,NYTreview,avgReview}\sigma_{year>1973 \wedge NYTreview \geq avgReview}(F_4)$ | | | | |
|---|---|---|---|---|---|---|---|
| ISBN | title | bestSeller | ISBN | title | year | NYTreview | avgReview |
| 001 | "Florida" | No | 003 | "Meditation" | 1974 | 8 | 7 |
| 002 | "China" | No | 004 | "Dreams" | 1980 | 9 | 3 |
| 003 | "Meditation" | No | 005 | "TCP/IP" | 1992 | 15 | 15 |
| 004 | "Dreams" | Yes | 006 | "HTML" | 1974 | 16 | 2 |
| 005 | "TCP/IP" | Yes | | | | | |
| 006 | "HTML" | Yes | | | | | |
| 007 | "Pens" | No | | | | | |
| 008 | "Quilts" | No | | | | | |
| 009 | "Micky" | No | | | | | |
| 010 | "Pooh" | No | | | | | |

| $R' = MJ(PID, L, F_1', F_2', F_3', F_4')$ | | | | | | |
|---|---|---|---|---|---|---|
| ISBN | title | year | oprahClub | bestSeller | NYTreview | avgReview |
| 002 | "China" | null | null | No | null | null |
| 003 | "Meditation" | 1974 | null | No | 8 | 7 |
| 004 | "Dreams" | 1980 | Yes | Yes | 9 | 3 |
| 005 | "TCP/IP" | 1992 | Yes | Yes | 15 | 15 |
| 006 | "HTML" | 1974 | null | Yes | 16 | 2 |
| 007 | "Pens" | null | null | No | null | null |
| 008 | "Quilts" | null | null | No | null | null |
| 009 | "Micky" | null | null | No | null | null |
| 010 | "Pooh" | null | null | No | null | null |

Figure 5.3: Compute CSET and content of $R'$ when $c_1$ = RandomEvidence or HighConfidence



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

| $F_1' = \pi_{ISBN,title,year,oprahClub}(F_1)$ | | | | $F_2' = \pi_{ISBN,oprahClub,bestSeller}(F_2)$ | | | |
|---|---|---|---|---|---|---|---|
| ISBN | title | year | oprahClub | ISBN | title | oprahClub | bestSeller |
| 001 | “Florida” | 1960 | No | 002 | “China” | No | Yes |
| 002 | “China” | 1966 | No | 003 | “Meditation” | No | Yes |
| 003 | “Meditation” | 1972 | Yes | 004 | “Dreams” | Yes | Yes |
| 004 | “Dreams” | 1970 | Yes | 005 | “TCP/IP” | Yes | Yes |

| $F_3' = \pi_{ISBN,title,bestSeller}(F_3)$ | | | $F_4' = \pi_{ISBN,title,year,NYTreview,avgReview}$ $\sigma_{year>1973 \wedge NYTreview \geq avgReview}(F_4)$ | | | | |
|---|---|---|---|---|---|---|---|
| ISBN | title | bestSeller | ISBN | title | year | NYTreview | avgReview |
| 001 | “Florida” | No | 003 | “Meditation” | 1974 | 8 | 7 |
| 002 | “China” | No | 004 | “Dreams” | 1980 | 9 | 3 |
| 003 | “Meditation” | No | 005 | “TCP/IP” | 1992 | 15 | 15 |
| 004 | “Dreams” | Yes | 006 | “HTML” | 1974 | 16 | 2 |
| 005 | “TCP/IP” | Yes | | | | | |
| 006 | “HTML” | Yes | | | | | |
| 007 | “Pens” | No | | | | | |
| 008 | “Quilts” | No | | | | | |
| 009 | “Micky” | No | | | | | |
| 010 | “Pooh” | No | | | | | |

| $R' = MJ(PID, L, F_1', F_2', F_3', F_4')$ | | | | | | |
|---|---|---|---|---|---|---|
| ISBN | title | year | oprahClub | bestSeller | NYTreview | avgReview |
| 001 | “Florida” | 1960 | No | null | null | null |
| 002 | “China” | 1966 | No | No | null | null |
| 002 | “China” | 1966 | No | Yes | null | null |
| 003 | “Meditation” | 1972 | Yes | Yes | 8 | 7 |
| 003 | “Meditation” | 1972 | Yes | No | 8 | 7 |
| 003 | “Meditation” | 1972 | No | Yes | 8 | 7 |
| 003 | “Meditation” | 1972 | No | No | 8 | 7 |
| 003 | “Meditation” | 1974 | Yes | Yes | 8 | 7 |
| 003 | “Meditation” | 1974 | Yes | No | 8 | 7 |
| 003 | “Meditation” | 1974 | No | Yes | 8 | 7 |
| 003 | “Meditation” | 1974 | No | No | 8 | 7 |
| 004 | “Dreams” | 1970 | Yes | Yes | 9 | 3 |
| 004 | “Dreams” | 1980 | Yes | Yes | 9 | 3 |
| 005 | “TCP/IP” | 1992 | Yes | Yes | 15 | 15 |
| 006 | ”HTML” | 1974 | null | Yes | 16 | 2 |
| 007 | ”Pens” | null | null | No | null | null |
| 008 | ”Quilts” | null | null | No | null | null |
| 009 | ”Micky” | null | null | No | null | null |
| 010 | ”Pooh” | null | null | No | null | null |

Figure 5.4: Compute CSET Phase for PossibleAtAll



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

| $c_1$ = HighConfidence | $c_1$ = RandomEvidence | $c_1$ = PossibleAtAll |
|---|---|---|
| $\delta_1 = <$ 004, "Dreams", 1970 $>$ | $\delta_1 = <$ 004, "Dreams", 1970 $>$ | $\delta_1 = \emptyset$ |
| $\delta_2 = \emptyset$ | $\delta_2 = \emptyset$ | $\delta_2 = \emptyset$ |
| $\delta_3 = \emptyset$ | $\delta_3 = \emptyset$ | $\delta_3 = \emptyset$ |
| $\delta_4 = \emptyset$ | $\delta_4 = \emptyset$ | $\delta_4 = \emptyset$ |

Figure 5.5: Value of $\delta_i$'s with $d$ = DISCARD

| | $c_1$ = HighConfidence | $c_1$ = RandomEvidence | $c_1$ = PossibleAtAll |
|---|---|---|---|
| $d$=ANY | <005,"TCP/IP",1992> | <004,"Dreams",1980>, <005,"TCP/IP",1992> | <003,"Meditation",1984>, <004,"Dreams".1980>. <005,"TCP/IP",1992> |
| $d$=DISCARD | <005,"TCP/IP",1992> | <005,"TCP/IP",1992> | <005,"TCP/IP",1992> |

Figure 5.6: Query Results

**Example 5.3.1** This example demonstrates using the algorithms given above to evaluate the following 6 queries that retrieve the title and year of publication of books published after 1973, and which are on both Oprah's Club list and the best seller list, or on none of them, and have a higher rating by the New York Times review than the average customer review.

**select** [*d*]*ISBN, title, year*
**from** *Books*
**where** *year* > 1973 **and** *oprahClub=bestSeller* **and** *NYTreview* $\geq$ *avgReview*
**with** $c_1$

where $c_1 \in$ {RandomEvidence, HighConfidence, PossibleAtAll} and $d \in$ {ANY. DISCARD}. Figure 5.3 shows the predicates pushed onto $F_i$'s to compute $F_i'$'s$(i = 1...4)$ in the case of *HighConfidence* and RandomEvidence. It also shows the result of the match join producing $R'$, from which one gets: $CSET$(*Books*, p, RandomEvidence) = $\{004, 005\}$. PID verification is performed based on this result. The $\delta_i'$*s* are computed when $c_1$ = HighConfidence or $d$ = DISCARD, shown in Figure 5.5. Based on this result: *CSET(Person, p,* HighConfidence*)* = {005}, Figure 5.4 shows the predicates pushed onto $F_i$s to compute $F_i'$s$(i = 1...4)$ in the case of $c_1$ = PossibleAtAll. It also shows the result of the match join producing $R'$, from which one gets: *CSET(Person, p,* PossibleAtAll*)* = $\{003, 004, 005\}$. Final results of the 6 queries are given in Figure 5.6. □



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

## 5.4 AURORA-RH Query Processing

As shown in the previous section, AURORA-RI sends subqueries to various sources to fetch part or all of fragments. These subqueries are posed against homogenizing views of data sources maintained by respective AURORA-RH mediators, which are responsible for processing the subqueries and shipping the results back to the AURORA-RI mediators.

Assume that the source database $B$ has been homogenized into the target view $H$. Let $Q$ be a relational query against $H$. AURORA-RH's query processor, AQP, translates this query into a set of queries over the source database, sends the queries for execution, and assembles the answer to $Q$, using the data returned. As shown in Figure 4.4, AQP consists of a query execution engine, a query rewriter, and a query optimizer.

### 5.4.1 AQP Query Execution Engine and QEPs

Ultimately, AQP executes a query based on a query execution plan (QEP) generated by the query rewriter and optimizer. Such QEPs are similar to the QEPs used by a query processor of a DBMS except they involve mediation enabling operators. The algebraic-based and cost-based manipulations of these QEPs will also be different from those in a traditional query processor. Algebraic-based manipulation of QEPs will be discussed later in this section. Cost-based manipulation of QEPs are not discussed in this thesis. However, cost-based mediator query optimization is an active area of research, as reviewed in Section 2.6.3.

In the context of AURORA-RH query processing, QEPs are expressions that involve only source relations - relations that reside in the underlying data source. A QEP can be depicted as an operation tree whose nodes are annotated with an operator name and an argument list. A non-leaf node of the tree is either an AURORA-RH primitive, *retrieve*, *rename*, *pad*, or *deriveAttr*, or a relational operator. The leaf nodes of the tree are source relations. The AQP query execution engine evaluates QEP trees bottom up.

### 5.4.2 Query Rewriting

In this section, mediator queries in the form of $\pi_L \sigma_p(M)$ are considered, where $L$ is a list of attributes in $M$ and $p$ is a predicate. The rewriting algorithm given below can be adapted for join queries. Via MAT-RH, the derivation of $M$ is captured as transformations, such as *RELmat* and *ATTRmat*, and domain mappings, such as Domain Structural Functions (DSFs) and Domain Value Functions (DVFs) in the View Definition Repository. The purposes of query rewriting are the following:

1. To modify a mediator query so that it only references source relations, not view relations.

2. To replace special transformations, such as *RELmat* and *ATTRmat*, with their definitions in



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

MEA-RH. For instance, the rewriting algorithm will replace

$$RELmat(D^R, A)$$

with

$$\bigcup_{i=1}^{n} pad(R_i, A, RELname(R_i))$$

The algorithm AQPrewriteQuery performs query rewriting.

**Algorithm. AQPrewriteQuery**
**Input:** $Q = \pi_L \sigma_p(M)$.
**Output:** A QEP for $Q$

Repeat until all the relations referenced in $Q$ are source relations:

1. Replace any derived relation reference by $Q$ with its definition stored in the View Definition Repository.
2. Replace *RELmat* and *ATTRmat* transformations with their definition.

∎

The above algorithm modifies a query expression repeatedly until all relations referenced are source relations, and all *RELmat* and *ATTRmat* transformations are replaced with the equivalent MEA-RH expressions. The resulting expression is a QEP.

To make the presentation cleaner, MEA-RH expressions are given as *operation trees*, rather than as long formulas. An operation tree is like a QEP except its leaf nodes may reference view relations. An operation tree that does not reference view relations is a QEP. Therefore, an operation tree can be rewritten into a QEP by repeatedly replacing the view relations with their derivations, as shown in the following example.

**Example 5.4.1** Consider query:

$$Q = \pi_{ISBN,title,price}\sigma_{price<45}\sigma_{category=\text{"Travel and Adventure"}}(Books)$$

that retrieves the *ISBN*, *title* and *price* of books of the category of "Travel and Adventure" that cost less than 45 US dollars. This query is posed against the homogenizing view as shown in Chapter 4. A graphical representation of this query is shown on the left of Figure 5.7. Rewriting of this query is performed as follows:

1. As shown in Example 4.2.8 of Section 4.2.5 , *Books* is a view relation with the following derivation:

   $$Books = deriveAttr(Books_v, \{price\}, price, f^v_{Books.price}, \{category\}, category, f^v_{Books.category})$$

   Replacing *Books* in $Q$ with the above expression gives $Q_1$, shown on the right of Figure 5.7.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

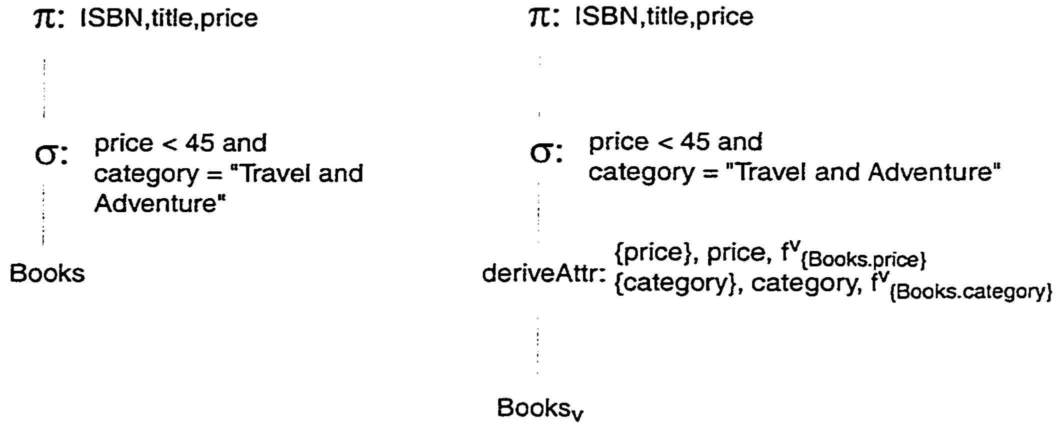



Figure 5.7: Query Rewriting: $Q_1$

2. As shown in Example 4.2.7 of Section 4.2.5, relation $Books_v$ is a view relation with the following derivation:

$$Books_v = \pi_{ISBN,title,category,bestSeller,price}(deriveAttr(Books_p, \{price, deduction\}, price, f^s_{Books.price}))$$

Replacing $Books_v$ in $Q_1$ with the above derivation gives $Q_2$, shown in Figure 5.8.

3. As shown in Example 4.2.4, relation $Books_p$ is a view relation with the following derivation:

$$Books_p = RELmat(D^R, category)$$

where $D^R = \{Travel, NewAge, Computer, Hobbies, Children\}$. Replacing $Books_p$ in $Q_2$ gives $Q_3$, shown in Figure 5.9. The transformation $RELmat$ in $Q_3$ must be replaced with its definition as given below:

$$\begin{aligned} RELmat(D^R, category) = \; & pad(Travel, category, \text{“Travel”}) \cup \\ & pad(Travel, category, \text{“NewAge”}) \cup \\ & pad(Travel, category, \text{“Computer”}) \cup \\ & pad(Travel, category, \text{“Hobbies”}) \cup \\ & pad(Travel, category, \text{“Children”}) \end{aligned}$$

Replacing the $RELmat$ transformation in $Q_3$ with the above definition gives $Q_4$, shown in Figure 5.10.

4. All the leaf nodes in $Q_4$ are still view relations, as discussed in Example 4.2.3 (Section 4.2.5). For instance, view relation $Travel$ has derivation

$$Travel = retrieve(Travel)$$

Replacing all view relations in $Q_4$ with their derivations gives $Q_5$, shown in Figure 5.11.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

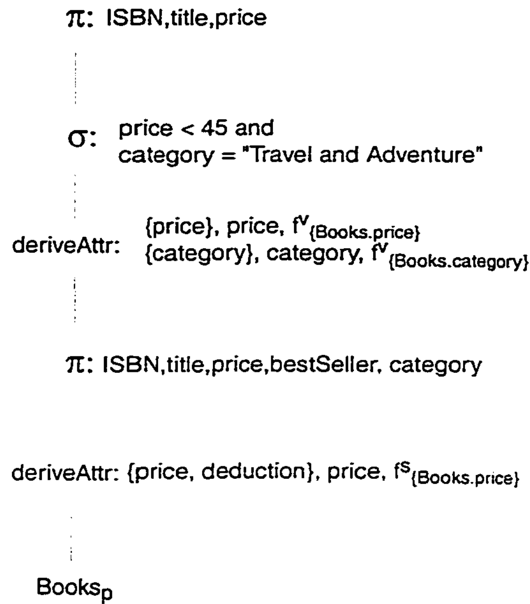



Figure 5.8: Query Rewriting: $Q_2$

$Q_5$ is a QEP since it only involves source relations and MEA-RH operators. □

### 5.4.3 AQP Query Optimization

The AQP query optimizer maximizes the number of relational operations performed by the source DBMS so as to leverage the query optimization capability of the source, and reduce the amount of data fetched into the AURORA-RH mediator. In a QEP, the *retrieve* nodes represent queries to be sent to the source DBMS for execution. The goal of query optimization in AURORA-RH is to transform the QEPs generated by query rewriting to *enlarge* the queries submitted to the source DBMS. As *retrieve* is the only operator that submits queries, the optimizer pushes as many as possible relational operators into *retrieve*. In order to achieve this goal, the AQP query optimizer performs two type of query modifications:

1. Relational operator push-downs. This type of modification pushes relational operators across MEA-RH operators towards the leaf nodes. Algebraic transformation rules are required for performing this modification.

2. Predicate modification. A relational operator can be pushed into *retrieve* if it is acceptable to the source query facility. Selections whose predicates involve functions that are not built-in in the source query facility do not exchange with *retrieve*. This potentially increases the amount of data fetched from the source. Predicate modification is a mechanism of eliminating



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

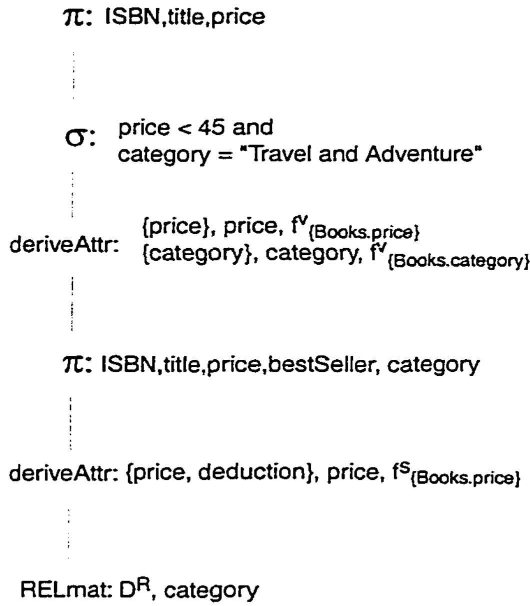



Figure 5.9: Query Rewriting: $Q_3$

user-defined functions from predicates so as to increase the chances of these predicates being pushed into a *retrieve* node. For example, if a predicate is in the form of

$$CDNtoUSD(price) > 45$$

and if the query processor "knows" that *CDNtoUSD* is a monotone increasing function with inverse $USDtoCDN$, it could rewrite this predicate into

$$price > USDtoCDN(45)$$

It can then evaluate the right hand side of the predicate to produce

$$price > 68$$

assuming that 45 US dollars is worth 68 Canadian dollars. This modified predicate can be pushed into a *retrieve* node easily.

Table 5.5 gives transformation rules for exchanging relational operators with *pad*, *rename*, and *deriveAttr*; these rules facilitate relational operator push-downs. For simplicity, the rules for *deriveAttr* are given only for cases where there is one derived attribute. Extensions can be easily made to allow multiple derived attributes. These rules are mostly self-explanatory. Proof of rules for *deriveAttr* is given in [96]. In Table 5.5, $p^{N \leftarrow X}$ denotes the predicate obtained from $p$ by substituting all appearances of $N$ with $X$. If $p$ does not involve $N$, $p^{N \leftarrow X} = p$. $L_{N \leftarrow A}$ denotes the list of attributes obtained from $L$ by replacing attribute $N$ with $A$. If $L$ does not involve $N$, $L_{N \leftarrow A} = L$.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

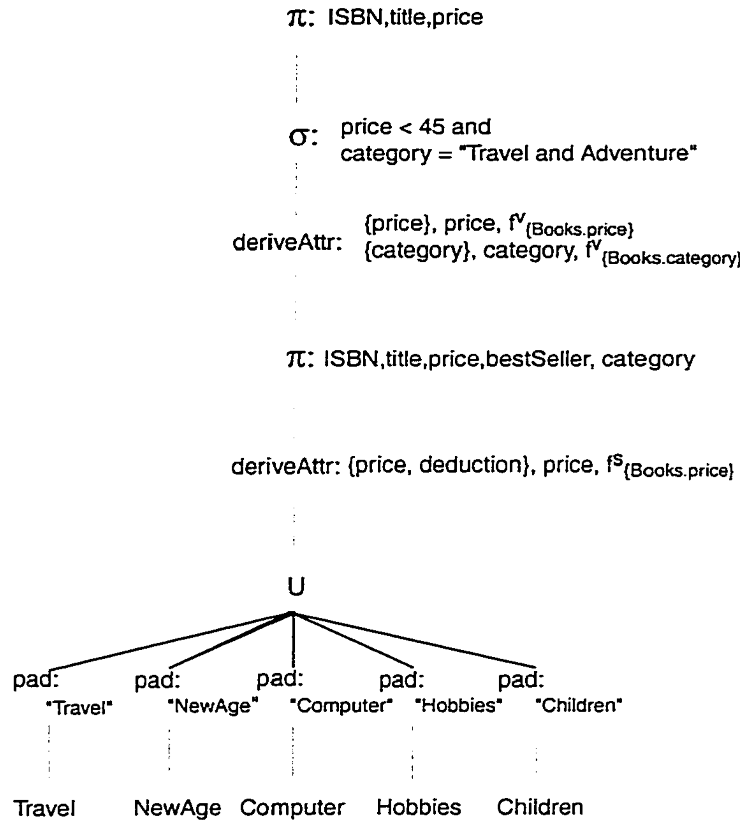



Figure 5.10: Query Rewriting: $Q_4$

A *control strategy* selects the next transformation rule to be applied. Currently AQP pushes relational operators across AURORA-RH primitives towards the leaves using the rules in Table 5.5, whenever and wherever applicable, in no particular sequence, until no rules are applicable. After each rule is applied successfully and if there are any changes to the predicates, AQP performs the predicate modification algorithm as given below. More sophisticated strategies to speed up optimization are a topic for future research.

**Algorithm.** PredicateModification ( $Q$ )
**input:** A QEP $Q$.
**output:** A modified QEP $Q$.

**BEGIN.**

Repeat until no modification can be made:



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

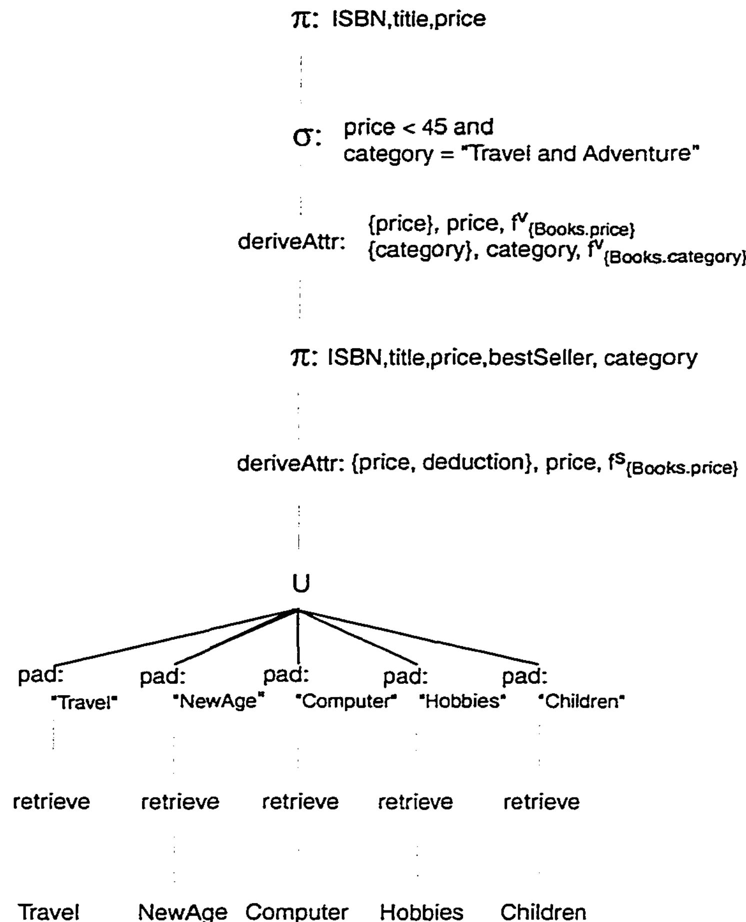



Figure 5.11: Query Rewriting: $Q_5$

For each subexpression in each predicate appearing in $Q$ in the form of $f(E_1)\theta f(E_2)$ or $f(E_3)\theta c$, where $E_1$, $E_2$ and $E_3$ are expressions, $c$ is a constant, $\theta \in \{=, >, <\}$, and $f$ is a function which has an inverse $f^{-1}$, if $f$ is strictly monotonic or $\theta$ is "=", replace this subexpression with $E_1 \theta E_2$ or $E_3 \theta c'$, respectively, where $c' = f^{-1}(c)$.

∎

**Example 5.4.2** This example is a walk-through of the optimization of the QEP shown in Figure 5.11.

1. First try to push the select, $\sigma$, operation near the top of the QEP tree across the *deriveAttr* operator beneath it, using rule $T_{deriveAttr}[3]$. This produces the QEP shown in Figure 5.12. Apply algorithm PredicateModification on this QEP, predicate

$$CNDtoUSD(price) < 45$$



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

| Transformation rules for *pad* | |
|---|---|
| $T_{pad}[1]$. | $\pi_L(pad(R, N, s)) \equiv \pi_L(R)$, $L \subseteq ATTR(R)$, $N \notin ATTR(R)$. |
| $T_{pad}[2]$. | $\pi_L(pad(R, N, s)) \equiv pad(\pi_{L-\{N\}}(R), N, s)$, $L \subseteq \{N\} \cup ATTR(R)$, $N \in L$. |
| $T_{pad}[3]$. | $\sigma_p(pad(R, N, s)) \equiv pad(\sigma_{p_{N \leftarrow s}}(R), N, s)$. |
| $T_{pad}[4]$. | $R \bowtie_p pad(R_1, N_1, s_1) \equiv pad(R \bowtie_{p_{N_1 \leftarrow s_1}} R_1, N_1, s_1)$. |
| $T_{pad}[5]$. | $pad(R_1, N_1, s_1) \bowtie_p pad(R_2, N_2, s_2) \equiv pad(pad(R_1 \bowtie_{p_{N_1 \leftarrow s_1, N_2 \leftarrow s_2}} R_2, N_1, s_1), N_2, s_2)$. |
| Transformation rules for *rename* | |
| $T_{rename}[1]$. | $\pi_L(rename(R, A, N)) \equiv \pi_L(R)$, $L \subseteq ATTR(R)$, $N \notin ATTR(R)$. |
| $T_{rename}[2]$. | $\pi_L(rename(R, A, N)) \equiv rename(\pi_{L_{N \leftarrow A}}(R), A, N)$, <br> $L \subseteq \{N\} \cup ATTR(R) - \{A\}$. |
| $T_{rename}[3]$. | $\sigma_p(rename(R, A, N)) \equiv rename(\sigma_{p_{N \leftarrow A}}(R), A, N)$. |
| $T_{rename}[4]$. | $R \bowtie_p rename(R_1, A_1, N_1) \equiv rename(R \bowtie_{p_{N_1 \leftarrow A_1}} R_1, A_1, N_1)$. |
| $T_{rename}[5]$. | $rename(R_1, A_1, N_1) \bowtie_p rename(R_2, A_2, N_2)$ <br> $\equiv rename(rename(R_1 \bowtie_{p_{N_1 \leftarrow A_1, N_2 \leftarrow A_2}} R_2, A_1, N_1), A_2, N_2)$. |
| Transformation rules for *deriveAttr* | |
| $T_{deriveAttr}[1]$. | $\pi_L(deriveAttr(R, L_1, N, f)) \equiv \pi_L(R)$, $L \subseteq ATTR(R)$, $N \notin ATTR(R)$. |
| $T_{deriveAttr}[2]$. | $\pi_L(deriveAttr(R, L_1, N, f)) \equiv \pi_L(deriveAttr(\pi_{L-\{N\} \cup L_1}(R), L_1, N, f))$, <br> $L \subseteq \{N\} \cup ATTR(R), N \in L$. |
| $T_{deriveAttr}[3]$. | $\sigma_p(deriveAttr(R, L, N, f)) \equiv deriveAttr(\sigma_{p_{N \leftarrow f(L)}}(R), L, N, f)$. |
| $T_{deriveAttr}[4]$. | $R \bowtie_p deriveAttr(R_1, L_1, N_1, f_1) \equiv deriveAttr(R \bowtie_{p_{N_1 \leftarrow f(L_1)}} R_1, L_1, N_1, f_1)$, <br> $ATTR(R) \cap ATTR(R_1) = \phi$, $N_1 \notin ATTR(R)$. |
| $T_{deriveAttr}[5]$. | $deriveAttr(R_1, L_1, N_1, f_1) \bowtie_p deriveAttr(R_2, L_2, N_2, f_2)$ <br> $\equiv deriveAttr(deriveAttr(R_1 \bowtie_{p_{N_1 \leftarrow f_1(L_1), N_2 \leftarrow f_2(L_2)}} R_2, L_1, N_1, f_1), L_2, N_2, f_2)$, <br> $ATTR(R_2) = \phi$, $N_1 \notin ATTR(R_2)$, $N_2 \notin ATTR(R_1)$, $N_1 \neq N_2$, $N_2 \notin L_2$. |

Table 5.5: Transformation Rules for *pad, rename* and *deriveAttr*

can be modified to

$$price < 68$$

and predicate

$$category = \text{“Travel and Adventure”}$$

can be modified to

$$category = \text{“Travel”}$$

Also, apply rule $T_{deriveAttr}[1]$ to the projection operator on top of the QEP and the *deriveAttr* beneath it, derivation of one of the attribute, *category*, can be eliminated. QEP after these modifications is shown in Figure 5.13.

2. Examine the $\sigma$ operator and push it across the projection beneath it and then try to push it across the *deriveAttr* operator lower in the tree, using rule $T_{deriveAttr}[3]$. Then push the projection on top of the tree across the *deriveAttr* beneath it and merge it with the projection operation under the *deriveAttr*. The QEP after these modifications is shown in Figure 5.14.

3. Push the predicate category = “Travel” across the $\cup$ operator and across the *pad* operator beneath it, using rule $T_{pad}[3]$. Many of the branches are eliminated. For instance,

$$\begin{aligned} & \sigma_{category=\text{“Travel”}}\, pad(Computer, category, \text{“Computer”}) \\ = \; & pad(\sigma_{\text{“Computer”}=\text{“Travel”}}\, Computer, category, \text{“Computer”}) \\ = \; & \emptyset \end{aligned}$$



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

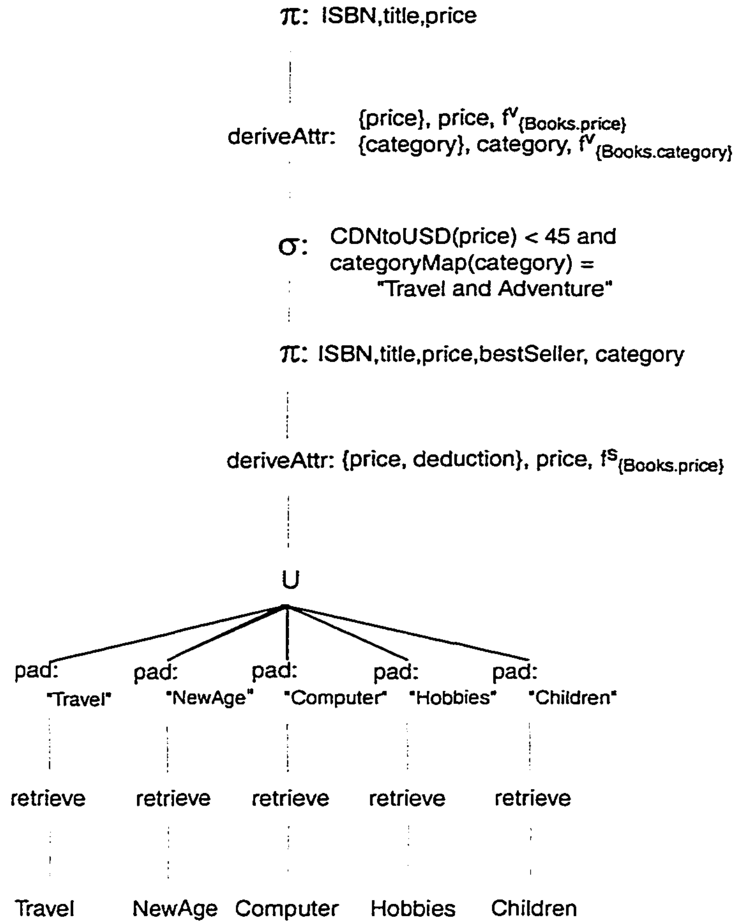



Figure 5.12: Query optimization example: first modification

The QEP after these modifications is shown in Figure 5.15.

4. Pushing the second projection operator down all the way to meet the *pad* operator and use rule $T_{pad}[1]$. The QEP after this modification is shown in Figure 5.16.

5. Finally, push the projection above the *retrieve* into it to get the final QEP, as shown in Figure 5.17.

The optimization has cut down the number of source relations involved from 5 to 1. □

## 5.5 Related Work

In [15, 21], algebraic rules for pushing selections across aggregation functions are studied under the assumption that schema integration is performed by an integration specification which resolves all potential instance level conflicts, using various aggregation functions. AURORA integration



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

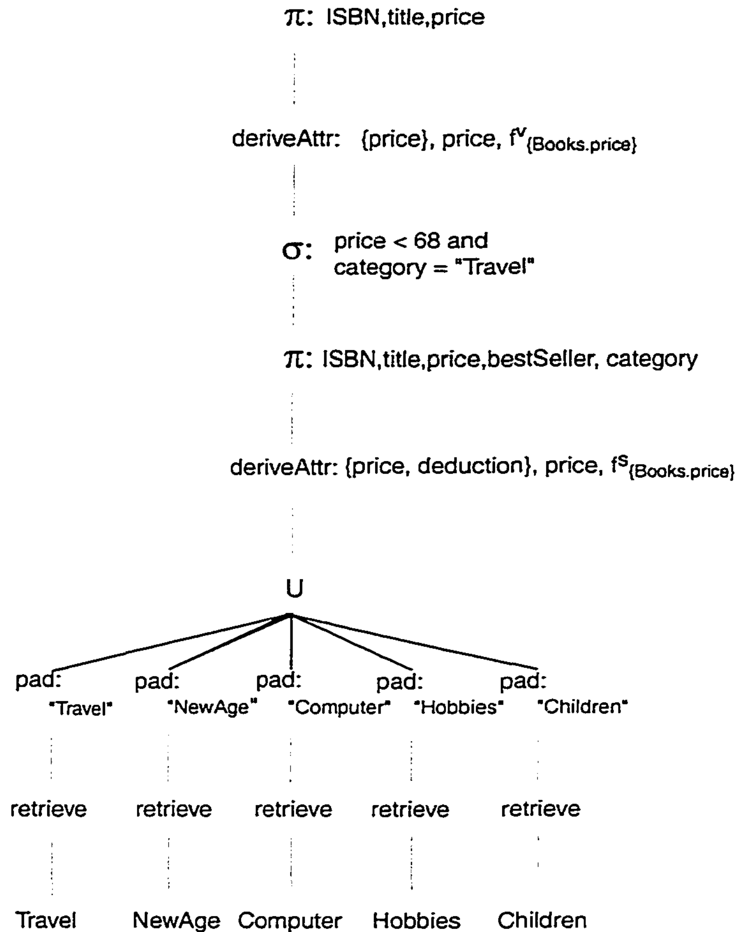



Figure 5.13: Query optimization example: second modification

mediators do not keep integration specifications; sources participate in the data service by registering with the mediator the data they can contribute. Conflicts are not resolved at schema integration time but rather tolerated at query time and resolved only upon returning of query results. In general, AURORA's approach towards instance level conflict handling offers a new way of querying potentially inconsistent data and new techniques for processing such queries efficiently.

The flexible relation model [23, 83] is designed to deal with instance level conflicts, but it requires the applications to use a non-standard data model for data access. This approach only deals with conflicts at predicate evaluation time and the tolerance mode is always *HighConfidence*. Conflicts in query results are not removed. Multiplex [68] deals with instance level conflicts in the context of answering queries using given materialized views. Conflicts arise when the materialized views overlap and the same query can be evaluated in multiple ways, resulting in multiple answers. A mechanism is proposed to derive an *approximate query answer* using these candidate answers. However, without any object matching assumption, it is not clear how conflicts can be detected.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

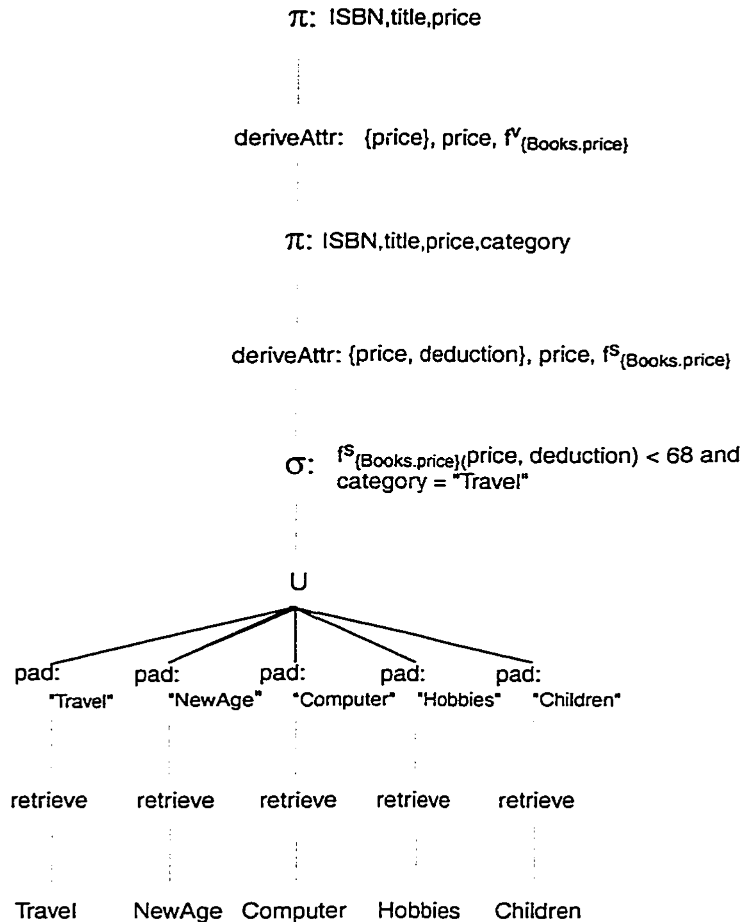



Figure 5.14: Query optimization example: third modification

[41] identifies domain mappings for resolving domain and schema mismatches. Resolutions for individual mismatches are demonstrated using an object-oriented database programming language. [41] does not provide a mediation methodology, nor does it explore query optimization techniques in the presence of the new language constructs. [42] provides a comprehensive classification of mismatches and conflicts. Resolutions for individual conflicts are given. New language constructs are proposed but query rewriting and optimization methods for these constructs are not given. [31] uses ontology to detect and resolve mismatches due to different units of measure. It is not clear how [31] handles other types of schematic mismatches.

Disco [89] extends ODMG ODL for mediation and proposes to use Volcano for query optimization. It introduces a logical operator *submit* and gives rules for exchanging relational operators with it. The cost model used is unclear. [25, 56] describe approaches that collect/establish statistics to build mediator query cost models. Query optimization in AURORA-RH focuses on single-source query modification techniques to leverage the source query optimization capability; a mediator query cost



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

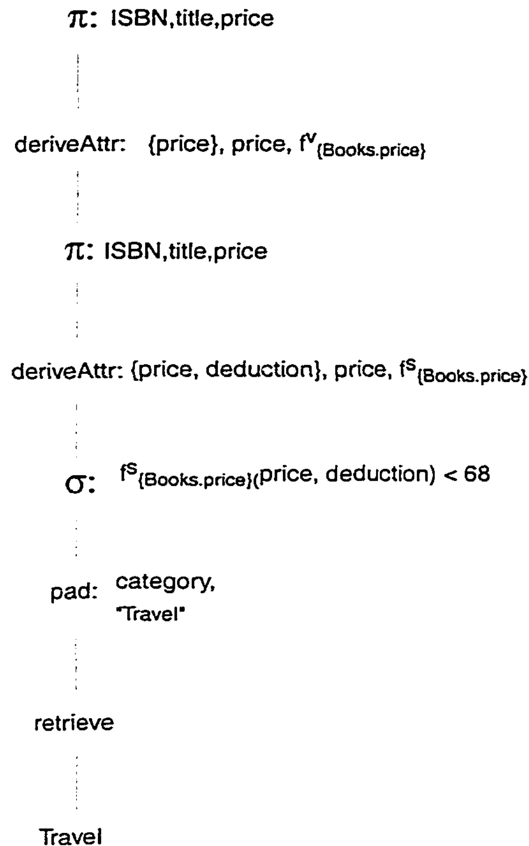



Figure 5.15: Query optimization example: fourth modification

model is not necessary. However, mediator query cost model is an interesting research topic.

## 5.6 Summary

The CT query model and related processing techniques are a new approach to handling instance level conflicts. Unlike previous approaches, conflicts are not resolved at schema integration time with aggregation functions, but are dealt with at query time. With the CT query model, instance level conflicts are tolerated to a degree acceptable to the applications. The advantage of this approach is that applications gain more control of the quality of the data access service they receive, and the mediators gain more room for query optimization. Techniques for optimized processing of CT queries have also been studied. In large scale data integration systems, the ability of optimizing query processing according to applications' requirements for data service quality is a significant factor in deployment.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

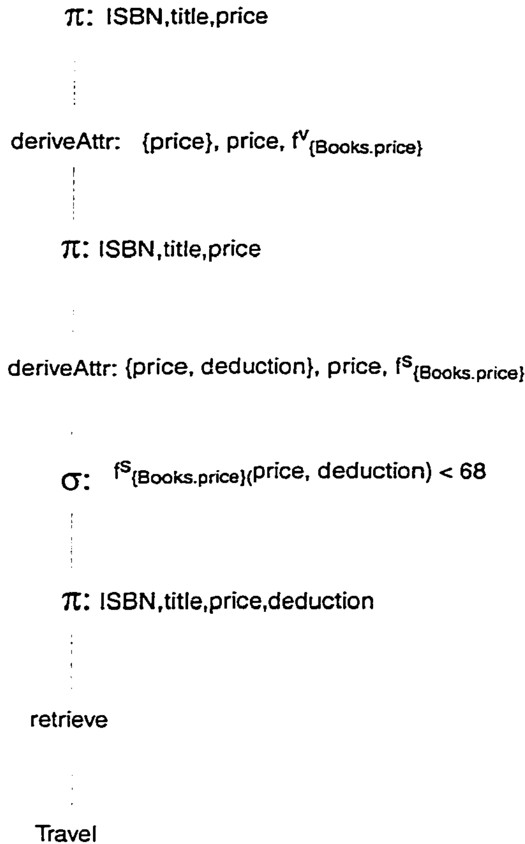



Figure 5.16: Query optimization example: fifth modification

AURORA-RH query processing and optimization are based on the MEA-RH algebraic framework, which allows the query processor to use knowledge gathered from the mediator author during homogenization, in order to build efficient QEPs. Homogenization is different from building a relational view in that it requires more sophisticated structural and semantic transformations of data. Fundamentally, MEA-RH allows the impact of the homogenization process on query processing efficiency to be studied. As demonstrated in this section, algebraic optimization of mediator queries can significantly reduce the volume of data retrieved into, and manipulated by, the mediator.

Future research in CT querying involves development of a cost model for strategy selection and a detailed performance study of the query optimization techniques presented here. Since query processing is a multi-phase procedure, apart from the major transformations that have been developed in this chapter, many smaller techniques for smart reuse of data retrieved in previous phases can be explored. These are engineering issues but may improve performance further.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

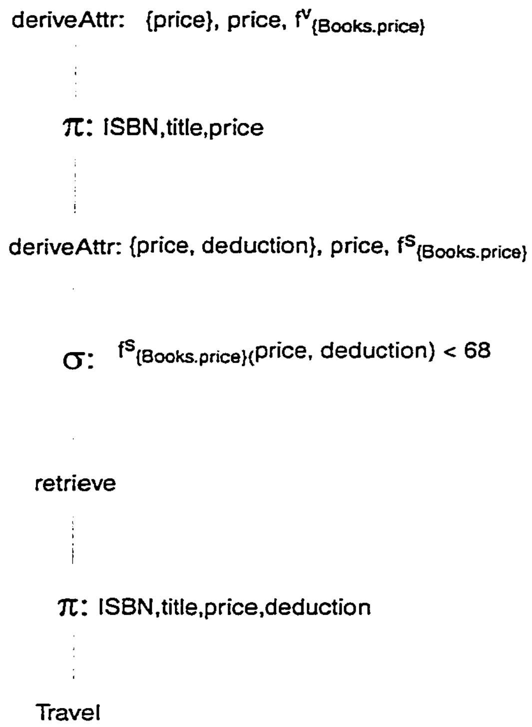



Figure 5.17: Query optimization example: sixth modification



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# Chapter 6

# Object-Oriented Mediation Framework

When the service view is based on an object data model, a mediation framework is needed for homogenizing and integrating data represented as objects that reside in various participating sources. This chapter describes AURORA's object-oriented mediation framework. As in the relational context, there are two mediators: object-oriented homogenization mediator, AURORA-OH, and object-oriented integration mediator, AURORA-OI.

## 6.1 The Service View

A service view is a schema written in ODMG ODL [14] that satisfies the following syntactical constraints:

1. Each object class must define a method with the following signature:

   $$PID : \emptyset \rightarrow T$$

   where $T$ is a Pure Literal Type (PLT). PID stands for **Plug-in Identifier**. Intuitively, the PID method returns a value that is required by the integration mediator for object-matching and oid generation; sources that wish to contribute data towards a global class in the service view must be able to perform the PID method of this class.

2. Class/interface specifications contain only methods: no attributes, relationships, types, constants, or exceptions.

3. Exports of parameters in methods are either *in* or *out*, not *inout*.

4. The parameter types and return types of methods are restricted to the following:



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(a) Pure literal types.

(b) Object types in the service schema.

(c) *set* $< T >$ where $T$ is an object type in the service schema.

This constraint says that the service view does not involve randomly complex types. This is because complicated structures with oids buried deep inside cannot be exported or interpreted easily; oids lose their validity once they leave the system where they are created. It is possible to devise a mechanism to exchange randomly complex values involving oids but this is not the focus of this research. This constraint is posed to simplify the initial development of the integration framework; it will eventually be removed.

These constraints do not restrict *what* can be represented in a service view; it only restricts *how* to represent them. For instance, a service view does not contain attributes but an attribute can be represented by a *get* method and a *set* method. Since the service view is read-only, a *get* method would represent an attribute completely. However, by restricting properties to methods, the integration mechanism of AURORA-OI is greatly simplified.

There are two ways of maintaining the extents of classes in the service view: as a materialized collection of oids; or as a virtual collection of oids. When dealing with a large number of sources that participate and withdraw from service views dynamically, materialized extents would be difficult to maintain. In AURORA, all extents in the service view are virtual, materialized only at run-time to entertain queries. It is the responsibility of the integration mediator to (partially) populate these virtual extents at run-time with *integrated objects* "manufactured" using objects exported by sources participating in the service view. Object classes in a service view are referred to as the *global classes*. Objects of global classes are referred to as the *global objects*.

**Notations.** The signature of a method defined on a class is in the following form:

$$N:\ e_1{:}T_1 \times \ldots \times e_n{:}T_n \rightarrow T_0$$

where $N$ is the method name, $T_0$ is the return type, $T_i$ and $e_i$ are the type and export of the $i$-th parameter, respectively, $e_i$ = *in/out/inout*. A *pure literal type* (PLT) is a literal data type that involves no object types; it is a type of pure value.

## 6.2 Data Sources and Wrappers

Wrappers must support an interface known to AURORA, as discussed earlier in Section 3.2.1. An AURORA-OH mediator must be able to access either ODMG sources or relational sources. To access a relational source, an AURORA-OH mediator must have the ability to "understand" data in the form of relations. Since AURORA-OH is based on the ODMG object model, relations can be viewed



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

as sets of "structs" and can be manipulated within the ODMG object model accordingly. This means that AURORA-OH mediators must provide a *mapper* which translates relations into sets of structs, and translates OQL queries over these sets of structs into SQL queries over relations. Construction of this mapper is an implementation issue and is beyond the scope of this dissertation. Generally, to participate in an object-oriented service view, data sources must be covered with a wrapper that facilitates accessing of the sources through an ODMG interface or a relational interface, whichever is most easily generated.

## 6.3 Overview of the Homogenization Framework

In the object-oriented context, a data source contributes data to a target service view by describing the data it provides as a set of *class fragments* of the global classes. Consider a service view containing three global classes: class *People* with methods *name()*, *age()*, and *phoneNumber()*; class *Dogs* with methods *name()*, *pedigree()* and *breed()*; and class *Cats* with method *longHaired()*. A class fragment of class *People* could be a class *SomePeople* defining methods *name()*, *phoneNumber()* and *address()*. A class fragment of class *Dogs* could be a class *SomeDogs* defining methods *name()* and *pedigree()*. A source may describe itself as being able to provide two classes, *SomePeople* and *SomeDogs*. Notice that this source does not contribute data towards *Cats*, nor does it contribute data on people's age or the breeds of dogs. Also notice that *SomePeople* is able to provide data on addresses of people although this information is not of interest to the service view. Nevertheless, *SomePeople* is a class fragment of *People*. Generally speaking, data sources contribute data on *some* aspects of *some* objects in a target service view.

### 6.3.1 The Homogenization Scenario

Homogenization is carried out by a *mediator author*, who designs an object-oriented homogenizing view. Some, usually not all, classes in this view are marked as class fragments of global classes: these classes are called the *homogenizing classes*. For instance, a homogenizing view of a data source may contain 5 classes arranged in some inheritance hierarchy. Among these classes are *SomePeople*, marked as a class fragment of global class *People*, and *SomeDogs*, marked as a class fragment of global class *Dogs*. A detailed discussion of homogenizing views is given in Section 6.3.2. Homogenizing views are derived by the mediator using an AURORA-OH mediator. Once the class fragments are explicitly marked, the homogenizing view and its relationship to the service view will be automatically understood by the target AURORA-OI integration mediator. This knowledge is used at run-time for combining source data to provide data to the applications.

In terms of semantics, a source class $S$ is a fragment of a global class $G$ if they describe the same application entity, although $S$ may describe some, not all, aspects that are of interest to $G$. In this



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

case, objects of $S$ and $G$ have the same *semantic intension*. Semantic intension of homogenizing classes in relation to global classes is determined by mediator authors; it is an important piece of semantic knowledge provided to the integration mediator. This knowledge is used by the integration mediator for object-matching - the process of identifying objects from various sources that describe the same application entity. Usually, several source objects describe the same global object, and these source objects are able to perform various methods; AURORA's integration mechanism is responsible for dispatching methods to the source objects that are able to perform them.

### 6.3.2 Homogenizing Views

It is the responsibility of the mediator author to design homogenizing views by determining which class fragments the underlying data source is able to provide, and which methods these class fragments are able to perform; these are semantic decisions that AURORA mediators do not automatically make - mediators only provide facilities for the mediator author to derive homogenizing classes as virtual classes. However, homogenizing view is a more relaxed notion than the usual object views [1, 81] for the following reasons:

1. Homogenizing classes may contain methods that are not related to the service view; these methods will not be used for integration purpose, but their presence does not impact on the data integration process. AURORA-OH does not provide facilities for "hiding" a method from a class. This makes type inferencing of derived classes much simpler.

2. Homogenizing view may contain classes that are not related to the service view; these classes will not be queried for integration purpose, but their presence does not impact on the data integration process. This also makes type inferencing of derived classes much simpler.

In general, AURORA-OH does not provide facilities to hide methods from classes or to hide classes from class hierarchies. The mediator author derives a homogenizing view and exposes some classes to the target integration mediator by marking these classes as fragments of respective global classes; these marked classes are the homogenizing classes. A method defined on a homogenizing class is exposed to the integration mediator only if its name and signature matches those of a method defined by $G$. These methods are referred to as the *export methods*. Methods that are not intended to be exposed to the integration mediator in this way must be renamed appropriately. AURORA-OH provides facilities for method renaming.

### 6.3.3 The Homogenization Facilities

Once the homogenizing view is designed, the mediator author uses an AURORA-OH mediator to derive it from the source schema by deriving the classes in the homogenizing view as virtual classes



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

on top of the source schema. In addition to the homogenizing classes, virtual classes are sometimes derived as an intermediate step towards the derivation of desirable homogenizing classes. AURORA-OH mediators, similar to their relational counterparts, provide *Mediation Enabling Operators* (**MEOs**) to facilitate these derivations. MEOs support the following types of derivations:

1. Regrouping of existing classes, either by taking the the union of existing classes or by selecting objects from an existing class based on a condition. For instance, one should be able to combine classes *Student* and *Employee* to form a new class *Person*. One should also be able to select all students older than 50 to form a new class *SeniorStudent*. With AURORA-OH, *Person* can be derived using the operator of generalization; *SeniorStudent* can be derived using the operator of specialization. These operators do not generate new objects; objects in *Person* or *SeniorStudent* already exist in the source. Such operators are said to be *object-preserving*.

2. Restructuring of existing classes. For example, one may want to derive a class *Specialist* with method *yearsInPractice()* and *specialization()* from class *Pediatrician*, defining only method *yearsInPractice()*. In this case, one should be able to declare a *virtual* method on class *Pediatrician*, *specialization()*, which returns a constant "Pediatrician". This operation restructures the *Pediatrician* class into a *Specialist* class by adding a new method to it. AURORA-OH provides an operator, *deriveOP*, for specifying virtual methods. This operator is object-preserving.

3. Merging/splitting of existing classes to generate new classes. For instance, one may want to derive the *Employee* class from a *Company* class that maintains an employee directory; this requires "splitting" a *Company* object into $x$ *Employee* objects where $x$ is the number of employees in the company. Or, one may need to derive *Department* objects from *Employee* objects, which requires merging of the *Employee* objects who work in the same department. Both of these operations can be expressed with MEO *OBJGEN* in AURORA-OH. In both cases, objects in the derived classes do not exist in the source but only exist in the homogenizing view; they are *imaginary objects*. Operators that produce imaginary objects are said to be *object-generating*. *OBJGEN* is the only object-generating operator in AURORA-OH.

4. Method mapping. Assume class *Employee* defines methods *salaryInCDN()*, *bonusInCDN()*, *manager()*, and *phoneNumber()*. Also assume that for homogenization purposes, a method *totalIncomeInUSD()*, which returns the total income of an employee including bonus, in US dollars, and a method *managerPhone()*, which returns the phone number of the employee's manager, must be derived. Method mapping should allow a mediator author to specify functions for converting Canadian dollars to US dollars and for deriving total income from salary and bonus. It should also allow specification of method *managerPhone()* as a path expression *Employee.manager().phoneNumber()*. In AURORA-OH, method mapping is supported by the operator of *deriveOP* which is an object-preserving operator.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The collection of homogenization MEOs provided by AURORA-OH overlaps with the operators previously proposed for constructing object views [1]. However, operators for hiding or importing of methods or classes are not needed. Rather, a more sophisticated mechanism for method mapping is required:

1. A framework needs to be defined for specifying, using user-defined functions, value conversions on in- and out- parameters, and the return values of methods. This framework should serve as a skeleton onto which the mediator author can "hang" her conversion functions so as to avoid reprogramming the methods themselves to incorporate such conversions. Consider the salary example again. The methods *salaryInCDN* and *bonusInCDN* may involve complicated calculations based on various business rules. To support method *totalIncomeInUSD*, the mediator author should not have to repeat these calculations. Instead, it should be possible to specify that *totalIncomeInUSD* is to be performed by an *Employee* object by performing *CDNtoUSD(SalaryAddBonus(THIS.salaryInCDN), THIS.bonusInCDN()))*. Therefore, all that mediator author has to do is to provide two conversion functions, *CDNtoUSD* and *SalaryAddBonus*, and to instruct the *Employee* objects on how to use these functions to perform *totalIncomeInUSD*.

2. A framework is required for declarative specification of virtual methods so as to exploit existing capabilities of objects and avoid the need to write new code. For instance, the mediator author should be able to specify *managerPhone* as a virtual method of *Employee* by specifying that each *Employee* object performs this method by performing *THIS.manager().phoneNumber()*.

Sources export information to the best of their capabilities. The only requirement is that if a source exports a class as a fragment of a global class, it must make sure that the class is able to perform the PID method defined by the global class. Currently, cases where a new method is coded are not considered; this facility will be added as future work.

### 6.3.4 Homogenization Methodology

AURORA-OH homogenization methodology and MEOs are closely related facilities. The homogenization methodology mandates that homogenization be performed in well-defined steps; in each step, only certain MEOs can be applied. A homogenizing class $C$ is derived in 2 steps: **object prototyping** followed by **method mapping**, as illustrated in Figure 6.1. Intuitively, object prototyping creates an object class $C_p$ that is *capable of* performing each of the methods defined on $C$; this capability is a semantic notion that only the mediator author understands. Moreover, there is a one-to-one mapping between $C_p$ objects and $C$ objects, such that the corresponding objects model the same application entity, with possibly different representations. $C_p$ is called a *prototype* of $C$. For instance, object class *Employee* with methods *employeeNo()*, *salaryInCDN()*, *bonusInCDN()*,



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

*manager()*, and *phoneNumber()* is a prototype of homogenizing class *Employee2* with methods *employeeNo()*, *totalIncomeInUSD()*, *managerPhone()*,..., because each *Employee* object corresponds to the *Employee2* object that has the same *Eno*, and is capable of performing all the methods defined by *Employee2* - although how exactly an *Employee* object performs these methods is yet to be specified. AURORA-OH supports object prototyping by providing facilities for regrouping, restructuring, merging, and splitting of object classes.

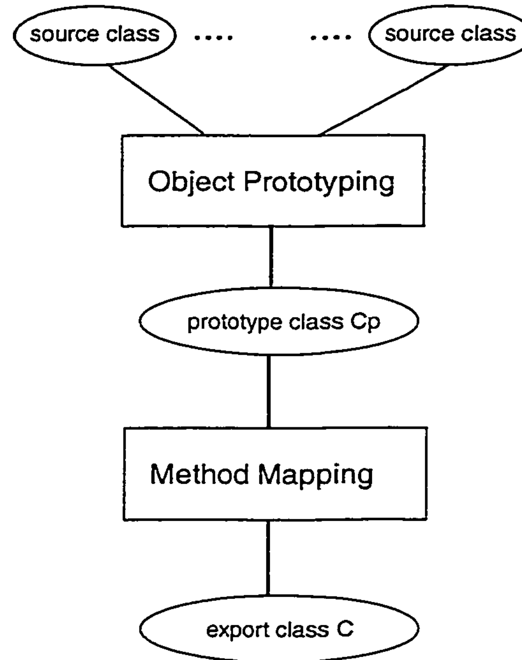



Figure 6.1: A Homogenization Methodology

Given any method $m$ defined by class $C$, a prototype class of $C$, $C_p$, must be able to perform $m$, that is, it is able to "do the same thing" as $m$; so far this ability is determined by and known only to the mediator author. The goal of method mapping, the second step of homogenization, is to allow the mediator author to express this semantics as a specification of exactly how $C_p$ objects perform $m$. To do so, the mediator author must derive $m$ as a *virtual method* of class $C_p$ by using the method mapping facilities provided by AURORA-OH. For instance, the mediator author can specify that an *Employee* object, $e$, performs *totalIncomeInUSD()* by invoking *CDNtoUSD(SalaryAddBonus(e.salaryInCDN(), e.bonusInCDN()))*, where *CDNtoUSD* and *SalaryAddBonus* are functions provided by the mediator author. Once method mapping is complete for every method defined on $C$, $C_p$ indeed becomes $C$. Thus the derivation of class $C$ is completed.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

### 6.3.5 AURORA-OH MEOs

AURORA-OH provides six Mediation Enabling Operators (MEOs):

#### Generalization

Generalization creates a virtual class as the superclass of existing classes. This is supported by the MEOs *Gen* and *BGen*. Operator *Gen* creates a class as the common superclass of a given set of classes. Operator *BGen* creates a class as the common superclass of a set of classes that support a given set of methods. These MEOs derive a new class and are in the following forms:

$$C_{new} = Gen(C_1, ..., C_n)$$

where $C_1, ..., C_n$ are existing classes; and

$$C_{new} = BGen(C_1, ..., C_n, S_I)$$

where $C_1, ..., C_n$ are existing classes and $S_I$ is a set of method signatures. For example, one can derive class *People* as a generalization of *Student* and *Professor* as follows:

$$People = Gen(Student, Professor)$$

Class *People*, thus derived, defines all the methods that are commonly defined by *Student* and *Professor*. The extent of *People* is the union of those of *Student* and *Professor*. One can also derive class *Printables* as a behavioral generalization of classes *AsciiFiles*, *Email*, *HtmlFile*, as follows:

$$Printable = BGen(AsciiFiles, Email, HtmlFiles, \{print()\})$$

Class *Printable*, thus derived, defines a single method *print()*. The extent of *Printable* is the union of all classes in the parameter list that define method *print()*.

#### Specialization

Specialization creates a virtual class as the subclass of an existing class by selecting existing objects in this class with an OQL query. This is supported by the MEO of *Spe*, in the following form:

$$C_{new} = Spe(C, Q)$$

where $C$ is an existing class and $Q$ is an OQL query that produce a set of objects in the extent of $C$. For instance, once can create class *SeniorStudent* from class *Student* as follows:

$$SeniorStudent = Spe(Student, \text{“}\mathbf{select}\ s\ \mathbf{from}\ students\ s\ \mathbf{where}\ s.Age() > 65\text{”})$$

#### Object Generation

This operation allows joining and splitting of existing classes to create new virtual classes. This is



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

supported by the MEO of $OBJGEN$, in the following form:

$$C_{new} = OBJGEN(Q, f)$$

where $Q$ is an OQL query that generates a set of tuples and $f$ is an *identifier function* that maps each element in the result of $Q$ to a unique value; the system then maintains a one-to-one mapping between this value and a unique oid. For instance, let $Q_1$ be the following:

$Q_1 =$ **select** DeptNo
**from** Employees e
**group by** DeptNo: e.DeptNo

and let $f_1$ be a function that maps an element in the result of $Q_1$, $v = < DeptNo : d, partition : x >$ to $d$, that is, $f_1(v) = d$, then the following operation

$$Department = OBJGEN(Q_1, f_1)$$

generates an object class *Department* that defines two methods: *DeptNo()* and *partition()*. For each distinct *DeptNo()* value of employees, $d$, a unique object of class *Department*, $o_d$, is created. Moreover, $o_d.DeptNo() = d$, $o_d.partition() = x$ if and only if $< d, x > \in Q$. For another example, consider object class *Company* that has a method *employeeDirectory* which returns a collection of tuples in the form of $< Eno : string, SIN : string, name : string, phoneNumber : string >$. To derive class *Employee*, *Company* objects must be "split". First, the following query is specified:

$Q_2 =$ **select** empData
**from** companies c, empData **in** c.employeeDirectory

and let $f_2$ be a function that maps a tuple $< Eno : e, SIN : s, name : n, phoneNumber : p >$ to $s$, that is, $f_2(< Eno : e, SIN : s, name : n, phoneNumber : p >) = s$. Then class *Employee* can be derived as follows:

$$Employee = OBJGEN(Q_2, f_2)$$

**Renaming methods**

This is achieved by the MEO of *Rename* in the following form:

$$Rename(C, N_{old}, N_{new})$$

The above operation renames the existing method $N_{old}$ in class $C$, and in all its subclasses, to $N_{new}$.

**Defining virtual methods**

AURORA-OH allows definition of virtual methods on virtual or base classes, and translations of parameters and return values of (virtual) methods using user-defined functions, path expressions, and OQL queries. This is supported by the MEO of *deriveOP*, that has the following form:

$$deriveOP(C, S_N, E, M_{in}, M_{out})$$



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This operator defines a virtual method on $C$ and all its subclasses. $S_N$ is the signature of the virtual method to be defined. $E$ is a *navigation method*. The concept of navigation method is a powerful one but for the current purpose it can be assumed to be a path-expression like specification. $M_{in}$ specifies in-parameter translation. $M_{out}$ specifies out-parameter and return value translation. These specifications, called *parameter translation maps*, allow mediator authors to specify value conversions using user-defined functions as well as path expressions. For example, the following operation derives method *managerPhone()* on class *Employee*:

$deriveOP(Employee, managerPhone, THIS.manager.phoneNumer,$
$\emptyset, \{G_0 = compactFormat(P_0)\})$

The above operation specifies that virtual method *managerPhone* is to be performed by an *Employee* object by invoking method *manager* on itself, and then invoking *phoneNumber* on the returned object. The last parameter says that the return value of *managerPhone* of an employee, *e*, is derived by translating *e.manager().phoneNumber()* from the format of "(123) 456-7890" to the format of 1234567890 using a user-provided function *compactFormat*. $G_0$ specifies that it is a conversion map for the return value, considered to be the 0-th out parameter. Generally, $M_{in}$ and $M_{out}$ provide the mediator author with a framework for "hanging" various conversion functions. As another example, the following operation derives method *totalIncomeInUSD* from *salaryInCDN* and *bonusInCDN*:

$deriveOP(Employee, totalIncomeInUSD, null, null,$
$\{G_0 = CDNtoUSD(SalaryAddBonus(THIS.salaryInCDN(), THIS.bonusInCDN()))\})$

### 6.3.6 A Compact Homogenization Example

In this section, a walk-through of an example is given to demonstrate the facilities AURORA-OH provides for homogenization. This example requires restructuring of object classes as well as generation of object classes whose extent contains imaginary objects. Assume that the class on the right is to be derived from the source class on the left:

```
class   Sales                                   class   ProductSales
        ( extent salesRec )                             (extent productSalesRec )
{                                               {
        short      month();           ==>               string       productType();
        integer    desktop();                           short        month();
        integer    laptop();                            integer      sales();
        integer    printer();                   };
};
```

The number of objects in class *ProductSales* is 3 times as great as that in class *Sales*; each *Sales* object must be split into 3 *ProductSales* objects. To derive class *ProductSales*, one first derives 3 classes: *desktopSales*, *laptopSales*, and *printerSales*, using the object generation operator $OBJGEN$. To derive class *desktopSales*, an OQL query is first specified:



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$Q_{desktopSales} = \textbf{select} \quad \textbf{struct}(salesObj{:}s,\ productType{:}\text{"desktop"}) \quad \textbf{from} \quad salesRec\ s$$

And let $f$ be an identity function on type $struct(salesObj : Sales, productType : string)$. Class *desktopSales* can then be derived by the following:

$$desktopSales = OBJGEN(Q_{desktopSales}, f)$$

The *desktopSales* class has two methods: $salesObj() : \emptyset \rightarrow Sales$ and $productType : \emptyset \rightarrow string$. Moreover, for any object $o$ in this class, $o.productType() =$ "*desktop*". The methods of *month* and *sales* on *desktopSales* are then derived as follows:

$deriveOP(desktopSales, month, THIS.salesObj.month, \emptyset, \{G_0 = P_0\})$

$deriveOP(desktopSales, sales, THIS.salesObj.desktop, \emptyset, \{G_0 = P_0\})$.

The first formula above says that to perform method *month*, a *desktopSales* object should first invoke method *salesObj* on itself, and then invoke method *month* on the returned result. The last parameter contains an *out-parameter translation map*, $G_0$, where $G$ indicates it is a translation map, and 0 indicates that the map is for the 0th out parameter of *month*, the return value. This map says that the return value of the new method *month* is the same as the 0th out-parameter ($P_0$) of *THIS.salesObj().month()*. The second *deriveOP* formula can be interpreted in a similar manner.

Similar to the specification of class *desktopSales*, classes *laptopSales* and *printerSales* can be derived. Once these classes are derived, the class of *ProductSales* can be derived as the generalization of these classes as follows:

$$ProductSales = Gen(desktopSales, laptopSales, printerSales)$$

Most operators used in the above example have their precursors in previously proposed constructs for object views [1], except the operator of *deriveOP*. Since the use of these operators has been illustrated in the above example, it will not be illustrated elaborately again but will be defined formally in AURORA terms.

The above example does not require complicated method mappings, for which AURORA-OH provides a richer and more elaborate framework than the object view systems [1]. Therefore, another example is designed as the running example to demonstrate the method mapping techniques of AURORA-OH. This example, described in Section 6.5.1, requires less regrouping, restructuring, and merge/split of objects, but requires sophisticated method mapping that will be carried out throughout the rest of this chapter.

### 6.3.7 MEOs and the Homogenization Methodology

All MEOs described can be used for object prototyping but only two MEOs can be used in the step of method mapping: *Rename* and *deriveOP*. When *Rename* and *deriveOP* are used for object-



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

prototyping, they facilitate restructuring of objects; when they are used for method mapping, they facilitate sophisticated translation of parameters, return values, and semantics of methods.

## 6.4 An Overview of the Object-Oriented Integration

In contrast to homogenization, which is a semi-automatic procedure that requires mediator authors' intervention and guidance in transforming individual sources, integration in AURORA is a fully automatic process dealing with a large number of participating sources through respective AURORA-OH mediators. The integration mediator, AURORA-OI, supports a fixed service view and is responsible for answering OQL queries posed against this view. To do this, AURORA-OI must be able to manufacture global objects using the objects provided by participating sources. This in turn requires AURORA-OI to do the following:

1. AURORA-OI must be able to access objects from multiple sources in a uniform manner.

   It is unlikely that AURORA-OI can access source objects using the oids assigned to these objects in their native sources. To use these oids as access handles, AURORA-OI must be linked with specific libraries and modules for each type of source. Integration would not scale, since access programs must be updated each time a new source is included. AURORA-OI uses run-time agents called *proxies* to access objects residing at various sources. A proxy represents one source object, but a source object may have any number of proxies. Proxies are the only type of handle used by AURORA-OI for accessing source objects; they are also responsible for facilitating the exchange of oids in a meaningful way across system boundaries, as parameters and as return values of methods. Proxies are generated at run-time by AURORA-OH mediators.

   In contrast, AURORA-OH mediators do not face the problem of dealing with objects from multiple sources as AURORA-OI mediators do. Generally, AURORA-OH is capable of manipulating objects using their source oids. This is why proxies are needed by AURORA-OI mediators, not AURORA-OH mediators. However, AURORA-OH mediators are where proxies are generated. All object exchanges between AURORA-OI and AURORA-OH must be done using proxies.

2. Given global class $C_g$, AURORA-OI must be able to combine objects from source classes registered as fragments of $C_g$ to create objects of $C_g$. Often multiple source objects represent various portions of the same global object; such source objects are identified by *object-matching.* A set of matching source objects, $S$, gives rise to a global object $o_S$; objects in $S$ are referred to as the *contributing source objects* of $o_S$. Each object in $S$ is capable of performing some, but usually not all, of the methods defined by $C_g$. $o_S$ performs any given method $m$ defined by $C_g$



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

by dispatching it to a source object in $S$ that is capable of performing it. Since source objects are represented by proxies, object-matching is performed using proxies. Global objects are manufactured by operator *Proxy Match Join* (PMJ), which matches proxies from all relevant sources to produce integrated objects.

Object-matching is a large issue by itself and is not a focus of research in AURORA; a simple matching assumption is employed: object-matching is based on PID values. Given source classes $S_1$ and $S_2$, both registered as fragments of a global class $C_g$, objects $o_1 \in S_1$ and $o_2 \in S_2$ are considered to be matching objects if they have the same PID value. This is why it is necessary that source classes wishing to contribute data to a global class be able to perform the PID method defined on this global class. PID values of global objects are the basis for maintaining unique and immutable oids for global objects.

## 6.5 Basic Concepts in Object-Oriented Homogenization

This section describes the basic concepts of AURORA's object-oriented homogenization framework. First, a running example is described. This example will be used for illustrating various concepts. The rest of the section describes two concepts that support the functioning of AURORA-OH: (1) an internal conceptual framework for describing object classes; and (2) the concept of *navigation methods*.

### 6.5.1 A Running Example

A service view is shown at the top of Table 6.1; it contains a class "Doctor" and its subclass "Specialist". These classes have related extents and support a set of methods. The rest of Table 6.1 shows the homogenizing views of three sources: $DS_1$, $DS_2$, and $DS_3$. These views are partial since only the homogenizing classes are shown. The global classes of which these homogenizing classes are fragments are indicated on the first line of the class specification. These homogenizing classes are to be derived by homogenizing the respective data sources. For convenience, the population of each of the homogenizing classes is also listed at the end of the class declaration. Table 6.2 shows the source schema at data sources $DS_1$, $DS_2$ and $DS_3$, respectively. These are pre-existing schemas. The population of each source class is also listed at the end of the declaration of each class.

### 6.5.2 A Framework for Describing Classes

To support the MEOs, AURORA-OH needs an internal framework for describing classes from the source, as well as those derived. This section describes this framework.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

| Service View | | |
|---|---|---|
| **class** | Doctor | ( **extent** doctors ) |
| { | | |
| | **string** | PID(); |
| | **string** | TelNo(); |
| | **string** | ClinicAddress(); |
| | **string** | Profile(); |
| | **set<string>** | PatientHistory( **in string** PatientID, |
| | | **out** Doctor PreviousDoc, **out date** FirstAppt); |
| }; | | |
| **class** | Specialist **extends** Doctor | ( **extent** specialists ) |
| { | | |
| | **string** | Specialization(); |
| | **short** | YearsAdvancedTraining(); |
| }; | | |
| Homogenizing View At $DS_1$ | | |
| **class** | MyDoctor | ( **extent** myDoctors ) *// fragment of: Doctor* |
| { | *// only methods of interest are listed.* | |
| | **string** | PID(); |
| | **string** | ClinicAddress(); |
| | **string** | Profile(); |
| }; | *// Population:* $o_{11}$*(001)"Smith",* $o_{12}$*(002)"Jones",* $o_{13}$*(003)"Hanks".* | |
| Homogenizing View At $DS_2$ | | |
| **class** | Pediatrician | ( **extent** pediatricians ) *// fragment of: Specialist* |
| { | | |
| | **string** | PID(); |
| | **string** | TelNo(); |
| | **string** | Profile(); |
| | **string** | Specialization(); |
| | **short** | YearsAdvancedTraining(); |
| | **set<ConsultationRecord>** | PatientHistory( **in string** PatientID, |
| | | **out** FamilyPhysician PreviousDoc, **out date** FirstAppt); |
| }; | *// Population:* $o_{21}$*(001)"Smith",* $o_{22}$*(004)"Low"* | |
| **class** | FamilyPhysician | ( **extent** familyDocs ) *// fragment of: Doctor* |
| { | | |
| | **string** | PID(); |
| | **string** | TelNo(); |
| | **string** | ClinicAddress(); |
| }; | *// Population:* $o_{23}$*(005)"Peters"* | |
| Homogenizing View At $DS_3$ | | |
| **class** | Orthopedics | ( **extent** docs ) *// fragment of: Specialist* |
| { | | |
| | **string** | PID(); |
| | **string** | TelNo(); |
| | **string** | Specialization(); |
| | **short** | YearsAdvancedTraining(); |
| | **set<string>** | PatientHistory( **in string** PatientID, |
| | | **out** Orthopedics PreviousDoc, **out date** FirstAppt); |
| }; | *// Population:* $o_{31}$*(002)"Jones",* $o_{32}$*(007)"Bond"* | |

Table 6.1: Example Global Schema and Source Homogenizing Views



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**Data Source $DS_1$**

```
class   MyDoctor                              (extent myDoctors ){
        string                                ID();
        Clinic                                clinic();
        String                                bio();
};      // Population: o11(001)"Smith", o12(002)"Jones", o13(003)"Hanks".

class   Clinic                                ( extent clinics ){
        string                                Name();
        string                                Address();
};
```

**Data Source $DS_2$**

```
class   Doctor                                (extent doctors ) {
        string                                PID();
        string                                Profile();
        Patient                               FindPatient(in string PatientID);
};
class   FamilyPhysician extends Doctor        (extent familyPhysicians ){
        string                                ClinicAddress();
};      // Population: o23(005)"Peters"

class   Pediatrician extends Doctor           (extent pediatricians ){
        string                                TelNo();
        string                                YearsOfTraining();
};      // Population: o21(001)"Smith", o22(004)"Low"

class   Patient                               ( extent patients ){
        string                                SIN();
        string                                GetConsultationRec(out Doctor LastDoc, out date FirstAppt);
};
```

**Data Source $DS_3$**

```
class   Orthopedics                           (extent orthopedics ){
        string                                PID();
        string                                TelNo();
        string                                Experience();
        short                                 search(in string SIN, out Patient patient, out date FirstAppt);
};      // Population: o31(002)"Jones", o32(007)"Bond"

class   Patient                               ( extent patients ){
        string                                SIN();
        string                                GetConsultationRec(out Doctor LastDoc);
};

class   Clinic                                ( extent clinics ) {
        Orthopedics                           SpecialistInvolved();
        string                                Address();
};
```

Table 6.2: Source schemas and populations



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

### Class Hierarchy in Homogenizing Views

The goal of homogenization is to derive a set of homogenizing classes that have the desirable extent, interface, and semantics for the methods defined on the interface. Each MEO that derives a new (virtual) class from given classes defines all these characteristics of the derived class, based on those of the operand classes. One of the important characteristics of object classes is their position in a class hierarchy. AURORA-OH does not keep a separate *view hierarchy,* as in [43]. Similar to [1], upon initialization, an AURORA-OH mediator imports the subclass hierarchy from the underlying ODMG source. This hierarchy is then modified as the mediator author derives new classes and methods using MEOs provided by AURORA-OH. Virtual classes are treated the same way as the classes originally defined in the data source. As described in Section 6.3.2, derivation of a homogenizing view does not require hiding methods from classes or hiding classes from the class hierarchy. This means that evolving the class hierarchy is much simpler in AURORA-OH than in general purpose object view systems such as those described in [1] and [81], where hiding of attributes, methods, and classes is required.

Homogenization generates a *homogenizing class hierarchy* in which the homogenizing classes are defined; this hierarchy is part of the homogenizing view. AURORA-OH does not export inheritance hierarchy semantics; it is considered to be a local semantics that is not of interest to the integration mediator. It is the responsibility of the AURORA-OH mediator to translate queries against the homogenizing view into queries against the source schema.

### Object classes

A distinction is made between *logical schemas* and *implementation schemas.* This distinction corresponds to ODMG's distinction between an ODL specification of a schema and the C++ header and sources generated by the ODL preprocessor. Both specifications describe OODB schemas but the former is on a logical level, while the latter is on an implementation level. A logical schema and its implementation schema are shown in Figure 6.2.

An implementation schema contains application classes as well as system classes. In ODMG databases, it is mandatory that all implementation classes be subclasses of a *root object class, d_Object,* from which application objects inherit many system-provided functions for manipulating database objects. Therefore, all database objects share *system characteristics* that are defined by the root object class. These characteristics of the objects are not of interest in the AURORA-OH framework. In contrast, logical schemas do not specify any system characteristics, or the object root as the ultimate superclass of all classes; it only describes the *logical characteristics* of object classes due to their application semantics, such as the interfaces of object classes, the semantics of each method in these interfaces, and the sub-class relationship among object classes. AURORA-OH describes objects on the logical level. As such, object classes may be shown without a superclass.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
// Logical Schema of Class Professor //
class Professor   ( extent professors )
{
      string               name();
      unsigned short       grant_tenure();
};


// Implementation Schema of Class Professor //
class Professor: public d_Object {
public:
      d_String             name();
      d_UShort             grant_tenure();
};

d_Extent<Professor>        professors;  // maintained by DBMS
```

Figure 6.2: Logical and Implementation Schemas of Class Professor

However, when these classes are implemented in an ODMG database, they'll be the subclass of the object root.

AURORA-OH views a class as a 4-tuple $C = <I, E, SUP, SUB>$. $C.I$ is a set of methods, called the *interface method set*, which includes *all* methods applicable to objects in class $C$, defined directly by $C$, or inherited from superclasses of $C$. $C.E$ is the extent of class $C$; it is the collection of oids of all objects in class $C$. $C.SUP$ is the set of immediate superclasses of $C$, and $C.SUB$ is the set of immediate subclasses of $C$. To define a class, all four aspects must be defined. In particular, each method in the interface method set must be described by a method signature and a description of the semantics, often given procedurally. Following the ODMG object model, methods in $C.I$ must have a unique name; overloading similar to function overloading in C++ is not considered. Moreover, when a class $X$ is a subclass of class $Y$, $Y.I \subseteq X.I$, $X.E \subseteq Y.E$.

Generally, a property that is an attribute or relationship can be viewed as a pair of methods: one that *gets* the value of the property and one that *sets* the value of the property. In AURORA-OH, the *gets* method in this pair is of interest and an attribute/relationship property is viewed as a 0-parameter method with the name of the property as method name, and the type of the property as the return type. In the rest of this chapter, only methods are discussed; all discussions apply to attributes and relationships.

### Virtual Classes and Imaginary Objects

Given an ODMG data source, all the classes that are already defined in this source are referred to as *base classes*. A *virtual class* is a class with a derived interface method set and a derived, virtual extent that can be (partially) materialized at query processing time. In AURORA-OH, virtual classes are



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

specified by MEOs that regroup and restructure previously defined classes. These operators are the subject of Sections 6.6.1 - 6.6.3. Once specified, a virtual class behaves exactly the same as a base class; it can be queried by OQL and can be used to derive other virtual classes.

A base class has an extent that is a materialized collection of oids. Virtual classes always have a virtual extent that is conceptually a collection of oids but the collection may not be physically maintained by AURORA mediators. Moreover, depending on how a virtual class is created, its extent may contain *real objects* or *imaginary objects* [1]. Real objects are objects that already exist in a data source; they have a unique and immutable oid that is maintained by the source where they reside. Imaginary objects exist only in the derived views, not in the data source. These objects are maintained by AURORA-OH to allow data values to be accessed like objects; they have unique and immutable oids, and they are materialized at run-time to entertain data access requirements.

### 6.5.3 Navigation Methods

*Navigation method* is a way of declaring virtual methods without writing code. Given a class $C$, the methods in $C.I$ are referred to as the *base methods* of $C$. Assume that method $p$ defined on class $C$ returns an object of class $C'$, which defines method $p'$. One can declare a new method on class $C$, $M_{new}$, by asking objects of class $C$ to first perform $p$ on themselves, get an object of $C'$ as the result, invoke method $p'$ on this object, and return the result of this last invocation to the client. This method can be declared as follows:

$$M_{new} = THIS.p.p'$$

$M_{new}$ is called a *navigation method.* For instance, *THIS.manager.phone* is a navigation method defined on class *Employee.*

Intuitively, a navigation method defined on a class provides objects in this class with a specification for *navigating* through the database to locate other objects and data values of interest. An object, $o$, can reach an object or value, $o'$, as the result of performing a method it is capable of performing. As such, navigation methods can be specified as *path expressions* which serve as a "map" for navigating. However, navigation methods in AURORA extend the usual concept of path expression in two ways:

1. Objects can locate objects and values of interests via the out-parameters of method invocations. This type of navigation is illustrated in Example 6.5.2.

2. Objects can locate objects and values of interest via *directed relationships* specified as ODMG OQL queries or user-defined functions and mapping tables.

In the rest of this section, the concepts of directed relationships and navigation methods are formally defined.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

DEFINITION **6.5.1** [Directed Relationships.] A *directed relationship* from object class $C_1$ to object class or ODMG literal type $T_2$ is a mapping $M$: $C_1 \rightarrow set < T_2 >$. Directed relationships exist in two forms in AURORA-OH:

1. $M = f$, where $f$ is a function $f{:}C_1 \rightarrow set < T_2 >$.

2. $M = R_{A_1 \rightarrow A_2}$ where $R$ is a (virtual) collection of structs of type $struct(A_1{:}C_1, A_2{:}T_2)$. $R_{A_1 \rightarrow A_2}$ is a mapping such that $R_{A_1 \rightarrow A_2}(o_1) = \{o_2 \mid < o_1, o_2 > \in R\}$.

A directed relationship is many-to-1 if $\forall o_1 \in C_1$, $|M(o_1)| \leq 1$, otherwise it is many-to-many. □

A directed relationship is a mapping that exists external to any object class, but is available to all classes for constructing navigation methods. This concept is illustrated by the following example.

**Example 6.5.1** Consider $DS_3$ shown in Table 6.2. An *Orthopedics* object does not "know" its clinic, a directed relationship can be specified to provide this missing link:

$Q =$ **select** struct(Doc:d, clinic:c)
**from** orthopedics d, clinics c
**where** d **in** c.SpecialistsInvolved()

$Q$ defines a virtual collection of structs of type *struct(Doc: Orthopedics, clinic: Clinic)* that link orthopedics with the clinics they are involved with. $Q_{Doc \rightarrow clinic}$ is a directed relationship from *Orthopedics* to *Clinic*. If a doctor works at only one clinic, then $Q_{Doc \rightarrow clinic}$ is many-to-1, otherwise it is many-to-many. □

A navigation method is constructed recursively, as described in Definition 6.5.2 below. Each step of recursion defines a move of navigating forward. There are four ways of moving forward:

1. By invoking a method and returning the return value of this invocation.

2. By invoking a method and returning an out-parameter used in this invocation.

3. By locating a relevant object/value through a many-to-1 directed relationship.

4. By locating a relevant set of object/value through a many-to-many directed relationship.

The following definition defines the signature of a navigation method thus constructed, and describes the semantics of a navigation method procedurally.

DEFINITION **6.5.2** [Navigation Method.] $THIS$ is a *navigation method* on class $C$ with signature $THIS : \emptyset \rightarrow C$. $\forall o \in C$, $o.THIS() = o$. If $X : e_1{:}T_1 \times ... \times e_m{:}T_m \rightarrow T_{return}$ is a navigation method on $C$, then the following are also navigation methods on $C$:



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. $D = X.P$, if $T_{return}$ is an object type and $P:\ e'_1{:}T'_1 \times ... \times e'_n{:}T'_n \rightarrow T_0$ is a base or navigation method defined on it. The signature of $D$ is:

$$X.P:\ e_1{:}T_1 \times ... \times e_m{:}T_m \times out{:}T_{return} \times e'_1{:}T'_1 \times ... \times e'_n{:}T'_n \rightarrow T_0$$

$\forall o \in$ C, $o$.D$(p_1, ..., p_m, p_{m+1}, p'_1, ...., p'_n)$ has the following semantics:

$$\{p_{m+1} = o.X(p_1, ..., p_m);\ \text{ return }\ p_{m+1}.P(p'_1, ..., p'_n);\}$$

2. $D = X.P[k]$, if $T_{return}$ is an object type and $P:\ e'_1{:}T'_1 \times ... \times e'_n{:}T'_n \rightarrow T_0$ is a base or navigation method defined on $T_{return}$, $1 \leq k \leq n$, $e'_k = out$ or $e'_k = inout$. The signature of $D$ is

$$X.P[k]:\ e_1{:}T_1 \times ... \times e_n{:}T_m \times out{:}T_{return} \times e'_1{:}T'_1 \times ... \times e'_n{:}T'_n \times out{:}T_0 \rightarrow T'_k$$

$\forall o \in$ C, $o$.D$(p_1, ..., p_m, p_{m+1}, p'_1, ...., p'_n, p'_{n+1})$ has the following semantics:

$$\{p_{m+1} = o.X(p_1, ..., p_m);\ p'_{n+1} = p_{m+1}.P(p'_1, ..., p'_m);\ \text{ return } p'_k;\}$$

3. $D = X.M$ where $M$ is a many-to-1 directed relationship from $T_{return}$ to $C_1$, the signature of $D$ is $X.M:\ e_1{:}T_1 \times ... \times e_n{:}T_m \times out{:}T_{return} \rightarrow C_1$. $\forall o \in C$, $o.D(p_1, ..., p_m, p_{m+1})$ has the following semantics:

$$\{p_{m+1} = o.X(p_1, ..., p_m);\ \text{return } element(M(p_{m+1}));\}$$

where *element* returns an element from a set value.

4. $D = X.M$ where $M$ is a many-to-many directed relationship from $T_{return}$ to $C_1$. The signature of $D$ is $X.M:\ e_1{:}T_1 \times ... \times e_n{:}T_m \times out{:}T_{return} \rightarrow set < C_1 >$. $\forall o \in C$, $o.D(p_1, ..., p_m, p_{m+1})$ has the following semantics:

$$\{p_{m+1} = o.X(p_1, ..., p_m);\ \text{ return }\ M(p_{m+1});\}$$

□

**Example 6.5.2** This example illustrates navigation methods using source $DS_2$ shown in Table 6.2:

- *THIS.FindPatient.GetConsultationRec* is a navigation method that returns a set of consultation records of a patient. This method has the following signature:

$$THIS.FindPatient.GetConsultationRec:\ in{:}string \times out{:}\text{Patient} \times out{:}\text{Doctor} \times out{:}date \rightarrow set < string >$$

- *THIS.FindPatient.GetConsultationRec[1]* is a navigation method that returns the previous doctor of a given patient. This method has the following signature:



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$THIS.FindPatient.GetConsultationRec[1]$ : *in:string* $\times$ *out*:Patient $\times$ *out*:Doctor $\times$ *out:date* $\times$ *out: set* $<$ *string* $> \rightarrow$ Doctor

Now consider source $DS_3$, shown in Table 6.2, and recall the directed relationship $Q_{Doc \rightarrow clinic}$ defined in Example 6.5.1. Assume that $Q_{Doc \rightarrow clinic}$ is many-to-1. $THIS.Q_{Doc \rightarrow clinic}$ is a navigation method on class *Orthopedics* with signature

$$\text{THIS}.Q_{Doc \rightarrow clinic} : \emptyset \rightarrow \text{Clinic}$$

This method returns the clinic an orthopedics is attached to. □

## 6.6 Mediation Enabling Operators for Homogenization

This section describes the mediation enabling operators provided in AURORA-OH for homogenization. These MEOs are used to derive virtual classes to remove semantic and structural differences between source classes and the global classes. These MEOs perform three types of transformations: regrouping, object generation, and method derivation.

### 6.6.1 Regrouping MEOs

Generalization, specialization, and behavioral generalization are three object-preserving MEOs. These MEOs create virtual classes whose populations consist of objects from existing base or virtual classes. Interface methods of the virtual classes created by these operators include some or all of the methods previously defined on the operand classes. These MEOs are similar in semantics to the view operators with the same name as defined in [1]. In this section, these operators are redefined in AURORA terms.

DEFINITION **6.6.1** [Generalization.] The generalization of object class $C_1, ..., C_n$

$$C = Gen(C_1, ..., C_n)$$

is defined as follows:

1. $C.I = C_1.I \cap ... \cap C_n.I$.
2. $C.E = C_1.E \cup ... \cup C_n.E$.
3. $C.SUP = D$, where $D$ is the *most specific common superclass* of $C_1, ..., C_n$.
4. $C.SUB = \{C_1, ..., C_n\}$.

□



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Similar to [1], the *most specific common superclass* of a given set of classes $C_1, ..., C_n$ is a class $D$ such that:

1. Class $D$ is an ancestor class of $C_1, ..., C_n$.

2. There exists no $D'$, a descendent class of $D$, such that $D'$ is an ancestor class of $C_1, ..., C_n$.

DEFINITION **6.6.2** [Specialization.] A specialization of object class C based on query $Q$, which returns a set of objects of class $C$,

$$C' = Spe(C, Q)$$

is defined as follows:

1. $C'$.I = C.I.

2. $C'.E = \{o \mid o \in C.E, p(o) = true\}$.

3. $C'.SUP = C$. $C'.SUB = \emptyset$.

□

DEFINITION **6.6.3** [Behavioral Generalization.] Let L = $\{C_1, ..., C_n\}$ be a set of object classes. Let P = $\{p_1, ..., p_m\}$ be a set of method signatures. $\forall i, j, 1 \leq i, j \leq n$, $P \subseteq C_i.I$, $P \subseteq C_j.I$, $C_i.E \cap C_j.E = \emptyset$, the behavioral generalization of classes L by P

$$C = BGen(L, P)$$

is defined as follows:

1. $C.I = P$.

2. $C.E = \{o \mid \exists i, 1 \leq i \leq n, o \in C_i.E, P \subseteq C_i.I\}$.

3. $C.SUP = \emptyset$. $C.SUB = \{C' \mid C' \in L, P \subseteq C'.I\}$.

□

### 6.6.2 The MEO for Object Generation: *OBJGEN*

Sometimes objects must be merged or split to form new objects during homogenization. To do this, an ODMG OQL query is first used to generate a set of values, called *data containers*, and then each of these containers is translated into an object of a virtual class. For example, to derive an object class *Family* from object class *Person*, a set of data containers, *family*, is first created:

*family* = **select** *struct(Wife: w, Husband: h)*
**from** *persons w*
**where** *w.sex() = "F"* **and** *w.husband() = h*



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Each element in *family* is a structure that holds all the "raw material" needed for generating *Family* objects. The MEO of $OBJGEN$ will do the rest of the work of generating class *Family* from *family*.

Objects generated by $OBJGEN$ are imaginary objects. Like other objects, they must have oids and must support a well-defined set of methods. $OBJGEN$ must assign oids to the imaginary objects it generates and also define the methods supported by them. Oid generation for imaginary objects is a rather controversial issue because such oids can only be assigned based on values, but an oid is by definition *not* value-based. AURORA's solution is described below.

The goal of $OBJGEN$ is to assign unique and immutable oids to data containers and turn them into objects. Uniqueness in this context can only be based on the assumption that there already exists a way of distinguishing one data container from another. This means that for a set of data containers, $S$, there exists an *identifier function*, $f$, an invertible function that maps a given data container $v_c$ to a value $f(v_c)$ that uniquely identifies $v_c$ from $S$. That is, there exists no $v_c' \in S$, $v_c' \neq v_c$, such that $f(v_c') = f(v_c)$. $f(v_c)$ is referred to as the *core value* of $v_c$. For example, the identifier function of *family* could be one that maps each element in *family*, $< w, h >$ to a value $< i_1, i_2 >$ where $i_1 = w.PID()$, and $i_2 = h.PID()$. Identifier functions must be provided by a mediator author; it is a way for the mediator author to describe the identification semantics of the data containers that have been created.

$OBJGEN$ maintains a one-to-one mapping between the core values of data containers and the oids assigned to the objects representing these data containers. Immutability of oids in this context requires that "an oid always represent the same data container". Since data containers are identified by their core values, a data container remains the same as long as its core value is not changed. By maintaining a one-to-one mapping between assigned oids and core values, $OBJGEN$ supports immutable oids. Change of core values gives rise to new imaginary objects and is equivalent to deleting an old imaginary object and adding a new one, with a new oid.

Implementation techniques may vary in the way these functions and mappings are generated and maintained. Conceptually, for each virtual class $C$ derived by $OBJGEN$, an invertible function, $OID_C$, as defined below, is maintained.

DEFINITION **6.6.4** [Function $OID_C$.] Let $C$ be a virtual class created from a set $S$ of data containers of type $T$, and let $f$ be an identifier function of $S$, $f : T \rightarrow T'$, where $T'$ is the type of the core values of the data containers in $S$. The OID function of class $C$, $OID_C$, is an invertible function with signature $OID_C : T' \rightarrow C$ such that, for any value $v$ of type $T'$, $OID_C(v)$ returns the oid assigned to the imaginary object generated from the value in $S$ that is uniquely identified by $v$. □

DEFINITION **6.6.5** [MEO $OBJGEN$.] Let $Q$ be an OQL query of type $T = struct(A_1{:}T_1, ..., A_n{:}T_n)$ and let $f$ be an identifier function on the result of $Q$, $f : T \rightarrow T'$, where $T'$ is the type of the core



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

value of the data containers in $Q$. Then

$$C = OBJGEN(Q, f)$$

creates a virtual class $C$ as follows:

1. $C.E = \{OID_C(I_v) \mid \exists v \in Q, I_v = f(v)\}$,

2. $C.I = \{PID, A_1, ..., A_n\}$. Method $A_i (1 \leq i \leq n)$ has signature $A_i : \emptyset \rightarrow T_i$. Method $PID$ has signature $PID : \emptyset \rightarrow T'$. $\forall o \in C.E$, $o.PID() = OID_C^{-1}(o)$, $o.A_i() = f^{-1}(OID_C^{-1}(o))[A_i]$, $1 \leq i \leq n$,

3. $C.SUP = \emptyset$. $C.SUB = \emptyset$.

□

**Example 6.6.1** Consider the set of data containers *family* specified earlier in this section. Let $f$ be a function that maps a given value *<Wife:w, Husband:h>* to a value *<WifeID:w.PIN(), HusbandID:h.PIN()>*. The following operation creates an imaginary class, *Family*:

$$Family = OBJGEN(family, f)$$

The extent of *Family* includes one object for each element of *family*. *Family* objects defines 3 methods: $PID()$, $Wife()$ and $Husband()$. Assume that function $OID_{Family}(v)$ is a program that creates oids by concatenating the name "Family" with each element in $v$. Then a struct $< Wife : w, Husband : h > \in family$ where $w.PIN()$ = "001" and $h.PIN()$ = "002" has oid "Family-001-002", and:

*Family-001-002.PID()* = <"001", "002">;
*Family-001-002.Wife()* = w;
*Family-001-002.Husband()* = h;

□

### 6.6.3 MEOs for Renaming and Deriving Methods

AURORA-OH supports derivation of a virtual method, $M_1$, as a "wrapper" method of an existing method, $M_0$, base or navigation. Once derived, $M_1$ works as follows:

1. Derive each in-parameter of $M_0$ from the in-parameters of $M_1$.

2. Invoke $M_0$ with the above derived in-parameters.

3. Derive each out-parameter and return value of $M_1$ from those returned from step 2, and return them.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It is the responsibility of the mediator author to specify the details of the derivations in steps 1 and 3, and to provide such specifications to the MEO of *deriveOP*, which facilitates the derivation of virtual methods. By doing so, *deriveOP* provides the mediator author with a framework for performing sophisticated parameter translation. AURORA-OH allows a mediator author to express these translations with three constructs:

1. *Parameter derivation expressions*, PDEs. A PDE is a function that derives a value from a given set of parameters using constants, user-defined conversion functions, as well as existing methods. For instance, a PDE that applies a user-defined function $f$ to the 1st parameter can be expressed as $f(P_1)$. PDEs are defined in Definition 6.6.7.

2. *Inward Parameter Translation Map*, IPTM. An IPTM is a set of PDEs, one for each in-parameter of $M_0$; these PDEs involve only the in-parameters of $M_1$. Intuitively, these PDEs are used to derive all the in-parameters of $M_0$ from the in-parameters of $M_1$.

3. *Outward Parameter Translation Map*, OPTM. An OPTM is a set of PDEs, one for each out-parameter of $M_1$ and one for the return value of $M_1$. Each of these PDEs can involve only the out-parameters and return value of $M_0$. Intuitively, these PDEs are used to derive the values of out-parameters and the return value of $M_0$.

If a class already has a method with the same name as a target virtual method but is semantically different from the latter, it must be renamed.

DEFINITION **6.6.6** [MEO *Rename.*] Let $C$ be an object class and let $N$ be a base or navigation method of $C$ that is not inherited from a superclass of $C$. The operator *Rename(C, N, N1)*, where *N1* is a name that is not used by any method defined on $C$ or any of $C$'s descendent classes, renames the method $N$ to $N1$ in $C$ and all its descendent classes. □

DEFINITION **6.6.7** [Parameter Derivation Expression (PDE).] Let $N: e_1{:}T_1 \times ... \times e_n{:}T_n \rightarrow T_0$ be a method defined on class $C$. Let $o$ be an object of $C$. Let $V = < v_0, ...., v_n >$ be a value of type $< T_0, ..., T_n >$. A *parameter derivation expression*(PDE) on $N$, $G$, its type, *type*$(G)$, its source parameter set, $SPset(G)$, and its evaluation based on $V$ and $o$, $EVAL(G, V, o)$, are defined recursively as follows:

1. $G = THIS$, is a PDE. *type(G)* $= C$, $SPset(G) = \emptyset$, $EVAL(G,V,o) = o$.

2. $G = c$, where $c$ is a constant, is a PDE. *type(G)* $=$ *type(c)*, $SPset(G) = \emptyset$, $EVAL(G,V,o) = c$.

3. $G = P_i$, $0 \leq i \leq n$, is a PDE. *type(G)* $= T_i$, $SPset(G) = \{P_i\}$, $EVAL(G,V,o) = v_i$.

4. If $G'$ is a PDE on $N$ of object type $T$, $p: e'_1{:}T'_1 \times ... \times e'_m{:}T'_m \rightarrow T'_0$ is a method defined on $T$, and $G_i$ is a PDE on $N$ of type $T'_i$ $(1 \leq i \leq m)$, then $G = G'.p(G_1, ..., G_m)$ is a PDE on



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$N$. *type(G)* $= T'_0$. *SPset(G)* $= SPset(G') \cup SPset(G_1) \cup ... \cup SPset(G_m)$, *EVAL(G, V, o)* $=$ *EVAL(G', V, o).p*$(EVAL(G_1, V, o), ..., EVAL(G_n, V, o))$.

5. If $G'$ is a PDE on $N$ of object type T, $p:\ e'_1{:}T'_1 \times ... \times e'_m{:}T'_m \rightarrow T'_0$ is a method defined on $T$, $e'_k = out$, and $G_i$ is a PDE on $N$ of type $T'_i$ $(1 \leq i \leq m)$, then $G = G'.p(G_1, ..., G_m)[k]$ is a PDE on $N$. *type(G)* $= T_k$. *SPset(G)* $= SPset(G') \cup SPset(G_1) \cup ... \cup SPset(G_m)$. *EVAL(G, V,o)* has the following semantics:

   $EVAL(G', V, o).p(EVAL(G_1, V, o), ...EVAL(G_{k-1}, V, o), t, EVAL(G_{k+1}, V, o), ..., EVAL(G_n, V, o))$;
   $EVAL(G, V, o) = t$;

6. Let $f:\ in{:}T'_1 \times ... \times in{:}T'_x \rightarrow T'_0$ be a function and let $G_1, ..., G_x$ be PDEs on $N$ of types $T'_1, ..., T'_x$, respectively. $G = f(G_1, ..., G_x)$ is a PDE on $N$. *type(G)* $= T'_0$, *SPset(G)* $= SPset(G_1) \cup ... \cup SPset(G_x)$, *EVAL(G, V, o)* $= f(EVAL(G_1, V, o), ..., EVAL(G_x, V, o))$.

□

Parameter mappings are specified as translation maps of two kinds: inward and outward.

DEFINITION **6.6.8** [Inward/Outward Parameter Translation Maps ( IPTM and OPTM )] Let

$$N:\ e_1{:}T_1 \times ... \times e_n{:}T_n \rightarrow T_0$$

and

$$D:\ e'_1{:}T'_1 \times ... \times e'_m{:}T'_m \rightarrow T'_0$$

be two methods where $N$ involves no *inout* parameters. The *inward parameter translation map* (IPTM) from $N$ to $D$ is a list of PDEs on $N$, $L_{in} = \{G_{in_1}, ...., G_{in_x}\}$, where $\forall v, 1 \leq v \leq x, 1 \leq in_v \leq m$, $\{e'_{in_1}, ..., e'_{in_x}\}$ are all the in/inout exports in $D$, *type*$(G_{in_v}) = T'_{in_v}$, *SPset*$(G_{in_v})$ contains only in-parameters in $N$. The *outward parameter translation map* ( OPTM ) from $D$ to $N$ is a list of PDEs on $D$, $L_{out} = \{G_{out_0}, ...., G_{out_y}\}$, where $out_0 = 0$, $\forall w, 1 \leq w \leq y, 1 \leq out_w \leq n$, $\{e_{out_1}, ..., e_{out_y}\}$ are all the out exports in $N$, *type*$(G_{out_w}) = T_{out_w}$, *SPset*$(G_{out_w})$ contains only return value and the out/inout parameters in $D$. □

**Example 6.6.2** Assume that an Employee class defines method FindSalary as follows:

| | | |
|---|---|---|
| *integer* | *FindSalary(* | *in string Year,* |
| | | *out integer salary, out integer bonus, out Employee manager)* |
| signature: | *FindSalary:* | *in:string × out:integer × out:integer × out:Employee → integer* |

Now consider ( virtual ) method Salary2:

| | |
|---|---|
| *integer* | *Salary2 ( in date Date, out string managerPhone)* |
| signature: | *Salary2: in:integer × out:string → integer* |



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that returns the salary of an Employee object. Also assume that this returned value is the sum of the salary and bonus an employee receives in a given year, in US dollars, while both salary and bonus returned as out-parameters in FindSalary are in Canadian dollars. The IPTM from *Salary*2 to *FindSalary* is the following:

$$L_{in} = \{G_{in_1}\},\ \ G_{in_1} = dateToYear(P_1)$$

where *dateToYear* is a function that takes an in-parameter of type **date** and returns the year in string form. Intuitively, this map specifies that there is only one in-parameter in *FindSalary*, the first parameter. This parameter to FindSalary, *in string Year*, is to be derived by applying the function *dateToYear* to the first parameter of *Salary2*. The OPTM from FindSalary to Salary2 is the following:

$$\begin{aligned} L_{out} &= \{G_{out_0}, G_{out_2}\}; \\ G_{out_0} &= CNDtoUSD(SalaryAddBonus(P_2, P_3)); \\ G_{out_2} &= P_4.phoneNumber(); \end{aligned}$$

The above map specifies that there are two out-parameters in *Salary*2, parameter 0 (the return value), and param-eter 2(*out string managerPhone*). The two PDEs, $G_{out_0}$ and $G_{out_2}$, specify how these two parameters are to be derived from the out-parameters of *FindSalary*. $G_{out_0}$ specifies that the return value of *Salary*2 is to be derived by applying function *SalaryAddBonus* to parameters 2 and 3 of *FindSalary*. $G_{out_2}$ specifies that the 2nd parameter of *Salary*2, *managerPhone*, is to be derived by taking the 4th parameter of *FindSalary* and invoking method *phoneNumber* on it. □

DEFINITION **6.6.9** [MEO *deriveOP*.] Let $C$ be an object class, $N : e_1{:}T_1 \times ... \times e_n{:}T_n \rightarrow T_0$ be a method involving no *inout* parameters, and $C$ or any of $C$'s descendent classes do not have a method named $N$. Let $D$ be a base or navigation method of $C$ with signature $D : e'_1{:}T'_1 \times ... \times e'_m{:}T'_m \rightarrow T'_0$. Let $L_{in} = \{G_{in_1}, ...., G_{in_x}\}$ be an IPTM from $N$ to $D$ and $L_{out} = \{G_{out_0}, ...., G_{out_y}\}$ be an OPTM from $D$ to $N$. The following operation

$$deriveOP(C, N, D, L_{in}, L_{out})$$

adds an interface method N to class $C$ and all its subclasses. For any object $o \in C.E$, the semantics of $o.N(p_1, ..., p_n)$ is defined procedurally as follows:

1. $\forall j, 1 \leq j \leq m$, let $a_j = EVAL(G_{in_j}, < p_1, ...., p_n >, o)$ if $e'_j = in$ or $e'_j = inout$, otherwise let $a_j = null$.
2. Let $a_0 = o.D\ (a_1, ..., a_m)$.
3. $\forall k, 0 \leq k \leq n$, if $k = 0$ or $e_k = out$, let $b_k = EVAL(G_{out_k}, < a_0, ..., a_m >, o)$. Return $b_0$.

□



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Operator *deriveOP* derives a virtual operation $N$ from a base or navigation method $D$ using user-provided parameter translation maps. The method $N$ thus derived is a "wrapper" method of $D$. $N$ works as follows: it first derives all the in-parameters of $D$ using the in-parameters of $N$ according to the IPTM, $L_{in}$, provided as the 4th parameter to *deriveOP*. It then invokes $D$ using the in-parameters just derived. Finally, it derives its own out-parameters and return values using the OPTM, $L_{out}$, provided as the 5th parameter of *deriveOP*. This operator is illustrated by the following example.

**Example 6.6.3** This example continues Example 6.6.2 to show how $Salary2$ can be derived from $FindSalary$. To add a virtual method $Salary2$ on class $Employee$:

$$deriveOP(Employee, THIS.FindSalary, L_{in}, L_{out})$$

For any employee $e$, the semantics of $e.Salary2(aDate, managerPH)$ is the following:

1. Let $a_1 = dateToYear(aDate)$;
2. Let $a_0 =$ *e.FindSalary(*$a_1$*, salary, bonus, manager)*;
3. $b_0 =$ *CNDtoUSD(SalaryAddBonus(salary, bonus)); managerPH = manager.phoneNumber(); return* $b_0$.

□

## 6.7 Homogenization with AURORA-OH

Homogenization is performed by mediator authors following a homogenization methodology. Each step prescribed by the methodology requires a mediator author to derive virtual classes that are closer to the target homogenizing classes in structure and semantics. These virtual classes are derived using the operators described in Section 6.6. This section first describes the mediation methodology of AURORA and then demonstrates how it can be used to homogenize the sources shown in Table 6.2 to generate respective homogenizing views in Table 6.1.

### 6.7.1 A Homogenization Methodology

A homogenizing class, $C$, is derived in 2 steps: **object prototyping** followed by **method mapping**. This process is illustrated in Figure 6.1. Object prototyping creates a *prototype object class* $C'$, that has the ability to perform each of the methods defined on $C$. This ability is determined by the mediator author. That is, it is the mediator author's responsibility to design and derive $C'$. However, there must be a one-to-one mapping between objects in $C'$ and those in $C$, in that the corresponding objects model the same application entity, with possibly different representations.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The goal of method mapping is for the mediator author to describe exactly how each method of $C$ can be performed by objects of $C'$. To do so, the mediator author must derive each method of $C$ as a ***virtual method*** of class $C'$. This step does not generate new classes or change the position of the prototype (virtual) class in the inheritance hierarchy; it defines a new, virtual method that is equivalent to a target method both syntactically and semantically. Once all the virtual methods are defined, one can simply rename $C'$ to $C$, that is, $C'$ becomes $C$.

The object-oriented homogenization methodology mandates the sequence in which transformations are performed. The transformations themselves are performed using AURORA-OH's Mediation Enabling Operators (MEOs), described in Section 6.6.

The object-oriented homogenization methodology is less elaborate than its relational counterpart; it is not based on a detailed classification of mismatches and how to remove them systematically. This is because the object-oriented data model is flexible and rich in semantics. An enumeration of all possible mismatches would be large. It will be difficult to design an approach to identify and resolve these mismatches systematically. In AURORA, it is expected that in an object-oriented context, most, if not all, mismatches encountered in a data integration process are up to the mediator authors to define, identify and resolve. AURORA provides a set of operators that are commonly recognized as useful for deriving object views - such as generalization, specialization and object generation - and provides a framework for deriving virtual methods using user-provided functions.

### 6.7.2 A Walk-Through of the Homogenization Example

This section gives a walk-through of the homogenization of $DS_1$, $DS_2$, and $DS_3$ as shown in Table 6.2 against the global schema, as shown in Table 6.1. The homogenizing views of the three sources are also shown in Table 6.1. As shown below, this example does not require step 1 of the mediation methodology to be performed. Generally, the object prototyping step requires regrouping, restructuring of existing object classes, and/or generation of object classes; it can be a giant step in many cases, but AURORA's mediation methodology does not provide fine granularity guidelines in this step. Various examples of the kind of transformations that could happen in this step are given in Section 6.3.6 and Section 6.6. The rest of this section contains a walk-through of the method-mapping step of the example shown in Tables 6.2 and 6.1.

At $DS_1$, class *MyDoctor* is already a prototype of the target class *MyDoctor*. In the step of method mapping, specify the following:

*deriveOP* *(MyDoctor, PID, ID,* $\emptyset$, $\{G_{out_0} = P_0\}$*)*
*deriveOP* *(MyDoctor, ClinicAddress, THIS.clinic.Address,* $\emptyset$, $\{G_{out_0} = P_0\}$*)*
*deriveOP* *(MyDoctor, Profile, bio,* $\emptyset$, $\{G_{out_0} = P_0\}$*)*

At $DS_2$, classes *FamilyPhysician* and *Pediatrician* are the prototypes of target classes with the same names in the export view shown in Table 6.1. In the step of method mapping, specify the



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

following:

| | |
|---|---|
| *deriveOP* | *(Pediatrician, YearsAdvancedTraining, YearsOfTraining,* $\emptyset$, $\{G_{out_0} = P_0\}$*)* |
| *deriveOP* | *(Pediatrician, Specialization, null,* $\emptyset$, $\{G_{out_0} =$ *"Pediatrician"*$\}$*)* |
| *deriveOP* | *(Doctor, PatientHistory, THIS.FindPatient.GetConsultationRecord,* $\{G_{in_1} = P_1\}$, $\{G_{out_0} = P_0, G_{out_2} = P_3, G_{out_3} = P_4\}$*)* |

Note that by specifying a virtual method *PatientHistory* on class *Doctor*, the same method is specified for both *Pediatrician* and *FamilyPhysician*. The signature of the navigation method *THIS.FindPatient.GetConsultationRecord* is given in Example 6.5.2. Also note the specification of method *Specialization*, where the third parameter is null. In this case, *deriveOP* adds a virtual method that returns a constant value, "Pediatrician".

At $DS_3$, class *Orthopedics* is a prototype of the class with the same name in the homogenizing view of $DS_3$. In the step of method mapping, specify the following:

| | |
|---|---|
| *deriveOP* | *(Orthopedics, YearsAdvancedTraining, Experience,* $\emptyset$, $\{G_{out_0} = P_0\}$*)* |
| *deriveOP* | *(Orthopedics, Specialization, null,* $\emptyset$, $\{G_{out_0} =$ *"Orthopedics"*$\}$*)* |
| *deriveOP* | *(Orthopedics, PatientHistory, THIS.search[2].GetConsultationRecord,* $\{G_{in_1} = P_1\}$, $\{G_{out_0} = P_0, G_{out_2} = P_5, G_{out_3} = P_3\}$*)* |

By Definition 6.5.2, the navigation method *THIS.search[2].GetConsultationRec* on *Orthopedics* has the following signature:

$$THIS.search[2].GetConsultationRec: \quad in{:}string \times out{:}Patient \times out{:}date \times out{:}short \times out{:}Doctor \rightarrow set < string >$$

It is easy to understand the last *deriveOP* formula above for specifying operation *PatientHistory* on *Orthopedics*.

## 6.8 AURORA-OI: The Integration Mediator

The AURORA-OI mediator supports a service view by manufacturing global objects using relevant objects exported by participating sources.

### 6.8.1 Oid Generation for Integrated Objects

Global objects manufactured by an AURORA-OI mediator are imaginary objects since they only exist in AURORA-OI, not in any data sources. Similar to the approach described in Section 6.6.2, AURORA-OI assigns unique and immutable oids to the generated objects by maintaining a one-to-one mapping between the assigned oids and the PID values. This mapping is captured by function *GOID*.

**Function** *GOID*. For each global class $C$ in the service view, AURORA-OI maintains an invertible function $GOID_C : T^C_{PID} \rightarrow C$, where $T^C_{PID}$ is the PID type of class $C$. For any PID



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

value of $C_g$, $v$, $GOID_C(v)$ is the oid assigned to the imaginary object in $C$ identified by PID value $v$. This function is similar to the function $OID$ defined in Definition 6.6.4. This function is named $GOID$ to emphasize that it is used for maintaining oids of global objects.

### 6.8.2 Fragments and Registrations

Object classes exported by various sources, through respective AURORA-OH mediators, are referred to as the *source classes*. Methods defined on these classes are referred to as the *source methods*. Source classes can be registered as *fragments* of global classes. Once registered, these classes are known to AURORA-OI as *registered fragments*. Semantic intension of the source classes, and the semantics of the source methods, must be maintained by mediator authors. It is the responsibility of the mediator authors working at various AURORA-OH mediators to guarantee that each source class registered as a fragment of a global class indeed models the "same" entity of interest as the global class, and that each source method with the same name as a global method "does the same thing" as the global method. AURORA-OI imposes a few syntactical conditions on source classes that are registered as fragments so that these classes can be interpreted correctly and automatically by AURORA-OI. The following definition describes the kind of data exchange that is possible.

Definition **6.8.1** [Portable and Applicable Types.] A source data type $T_s$ is *portable* to a global data type $T_g$ if

1. Both $T_s$ and $T_g$ are pure literal types and $T_s = T_g$; or

2. $T_s$ is a registered fragment of $T_g$ or of a subclass, directly or indirectly, of $T_g$; or

3. $T_s = set < T_s' >$, $T_g = set < T_g' >$, and $T_s'$ is portable to $T_g'$.

A global data type $T_g$ is *applicable* to a source data type $T_s$ if

1. Both $T_s$ and $T_s$ are pure literal types and $T_s = T_g$; or

2. $T_s$ is a registered fragment of $T_g$ or of a superclass, directly or indirectly, of $T_g$; or

3. $T_s = set < T_s' >$, $T_g = set < T_g' >$, and $T_g'$ is applicable to $T_s'$.

□

There are two directions that data can be passed: from AURORA-OH to AURORA-OI and from AURORA-OI to AURORA-OH; different types of data can be passed in each direction so that the data passed into an mediator can be interpreted properly. To distinguish the two directions in which data are passed, AURORA-OH mediators are said to *port* data to AURORA-OI mediators and the latter *apply* data to the former. The concepts of portable types and applicable types then define



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the kind of data passing that is possible in the two directions. AURORA-OH and AURORA-OI can only exchange data values using portable types and applicable types.

If a source data type $T_s$ is portable to a global data type $T_g$, then a value of type $T_s$ can be passed by AURORA-OH to AURORA-OI, and this value can be interpreted by AURORA-OI as a type $T_g$ value. According to the above definition, $T_s$ and $T_g$ must satisfy one of the following conditions: (1) they are identical PLTs; or (2) the semantic intension of $T_s$ is the same as that of $T_g$, in which case $T_s$ is a registered fragment of $T_g$; or it is a specialization of the semantic intension of $T_g$, in which case $T_s$ is a registered fragment of a descendent class of $T_g$. Intuitively, $T_s$ represents an entity that is by semantics *a kind of* $T_g$.

Similarly, if a global data type $T_g$ is applicable to a source data type $T_s$, then a value of type $T_g$ can be passed by AURORA-OI to AURORA-OH and this value will be interpreted by the latter as a value of type $T_s$. According to the above definition, $T_s$ and $T_g$ must satisfy one of the following conditions: (1) they are identical PLTs; or (2) The semantic intension of $T_g$ is the same as that of $T_s$, in which case $T_s$ is a registered fragment of $T_g$; or it is a specialization of the semantic intension of $T_s$, in which case $T_s$ is a registered fragment of an ancestor class of $T_g$ in the service view. Intuitively, $T_g$ represents an entity that is by nature *a kind of* $T_s$.

DEFINITION **6.8.2** [Delegatable method.] A source method with signature $N: e_1{:}T_1 \times ... \times e_n{:}T_n \rightarrow T_0$ is a *delegatable method* of a global method $N_g : e_1^g{:}T_1^g \times ... \times e_m^g{:}T_m^g \rightarrow T_0^g$ if:

1. Method $N_g$ has the same name as method $N$ and $m = n$.

2. $T_0$ is portable to $T_0^g$. $\forall i, 1 \leq i \leq n$, $e_i = e_i^g$. Moreover, if $e_i = in$, then $T_i^g$ is applicable to $T_i$, if $e_i = out$, then $T_i$ is portable to $T_i^g$.

□

A delegatable method of a given global method is a source method whose signature *qualifies* to execute the global method; it is the mediator authors' responsibility to ensure that the source method qualifies in semantics as well. In Table 6.1, *PatientHistory* in classes *Pediatrician* and *Orthopedics* are both delegatable methods of the global method *PatientHistory* of *Doctor*.

DEFINITION **6.8.3** [Valid Fragments.]. A source class $C_f$ is a *valid fragment* of global class $C_g$ if

1. $C_f$ has a PID method that is identical in signature to that of $C_g$'s; and

2. $\exists X$, a set of methods defined on $C_g$, such that $\forall f_g \in S$, $\exists f$ defined on $C_f$ such that $f$ is a delegatable method of $f_g$.

□



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This definition says that a valid fragment of a global class must support the PID method of the global class and should define delegatable methods for *some* of the methods defined in the global class. In Table 6.1, source classes *FamilyPhysician*, *Pediatrician* and *Orthopedics* are valid fragments of global classes *Doctor* and *Specialist*.

A source class $C$ can be registered as a fragment of global class $C_g$ only if $C$ is a valid fragment of $C_g$, but not all valid fragments are registered fragments: which source class to register with which global class is determined by the mediator authors at sources. For instance, the mediator authors may decide to register *FamilyPhysician* with *Doctor*, *Pediatrician* and *Orthopedics* with *Specialist*, and so on. A registered fragment of $C_g$ is said to be ***compatible*** with $C_g$ and all its super-classes. Methods are registered implicitly. If $C$ is a registered fragment of $C_g$, then a method of $C$ that has the same name as a method of $C_g$ is a ***registered method***. It is the mediator authors' responsibility to make sure that all registered methods are delegatable methods of the global methods they share the same names with. All methods listed in the homogenizing views in Table 6.1 are registered methods.

### 6.8.3 Proxies

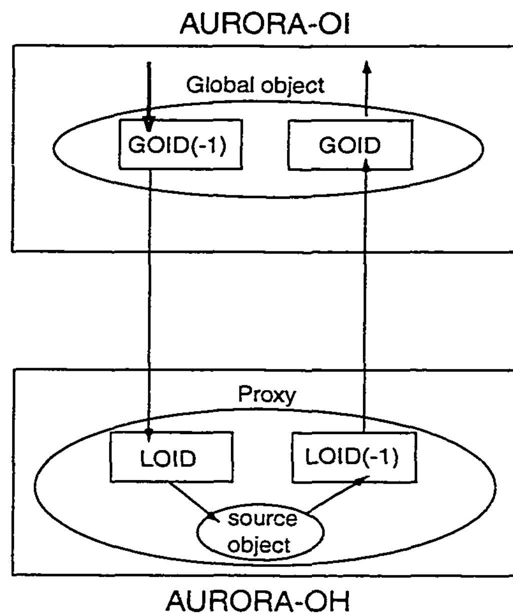



Figure 6.3: Use of Proxy for Uniform Access

AURORA uses *proxies*, handles generated by various AURORA-OH mediators to enable uniform access of source objects. Each proxy represents one source object but a source object may have any number of proxies. Proxies exist to perform the following tasks:

1. To accept requests from a foreign client, pass this request to the source object it represents



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

for execution, and return the results to the client.

2. To facilitate passing oids across system boundaries. Methods on local objects often take oids as in-parameters and return oids. Proxies must facilitate such exchange of oids in a meaningful way.

Uniform access of source object using proxies is illustrated in Figure 6.3. Proxies run at various AURORA-OH mediators and they accept PID values in place of oids as in-parameters for method invocation. A proxy executes a method by translating the PID values passed as in-parameters into local oids using a system function $LOID$, and uses the obtained oids as in-parameters to invoke the corresponding method on the source object it represents. Upon returning of this invocation, the proxy translates all returned oids, either as return value or as out parameters, into appropriate PID values using the inverse of function $LOID$, and returns the obtained PID values to the client. AURORA-OI accesses source objects through proxies. Global objects at AURORA-OI invoke a source method by first translating any in-parameters that are oids into PID values, using the inverse of function $GOID$. Upon returning of a request to a proxy, the global object translates the returned PID values into oids, using function $GOID$. Function $GOID$ was defined in Section 6.8.1. Function $LOID$ is defined below.

**Function** $LOID_C$. For each class $C$ registered, the AURORA-OH mediator where this class resides must maintain a function $LOID_C : T^C_{PID} \rightarrow C$, where $T^C_{PID}$ is the PID type of $C$, that maps a given PID value of $C$ to the $C$ object identified by $v$. That is, given PID value of class $C$, $p$, $LOID_C(p) = o$ if and only if $o.PID() = v$.

Proxies of different source objects are able to handle different requests; this is defined by proxy types. The semantics of methods on proxies is defined in Definition 6.8.5.

DEFINITION **6.8.4** [Proxy Type.] Let $C$ be a registered fragment and $F$ be all the registered methods of $C$. The proxy type of $C$, $PXY(C)$, is defined as follows: $\forall p \in F$ with signature $N : e_1{:}T_1 \times ... \times e_n{:}T_n \rightarrow T_0$, $PXY(C)$ defines a method $p'$ with signature $N : e_1{:}T'_1 \times ... \times e_n{:}T'_n \rightarrow T'_0$ where $\forall i, 0 \leq i \leq n$, if $T_i$ is a pure literal type, $T'_i = T_i$. If $T_i$ is an object type $C'$ or $set < C' >$, then $T'_i = T_{pid}$ or $T'_i = set < T_{pid} >$ respectively, where $T_{pid}$ is the PID type of object class $C'$. PXY(C) does not define any other methods. □

$PXY(C)$ defines a *proxy method* for each registered method of $C$. The signatures of the proxy methods are modified from the original signature to accept and/or return PIDs in place of oids.

DEFINITION **6.8.5** [Proxies.] Let $o$ be an object of a registered fragment $C$ and let $F = \{f_1, ..., f_n\}$ be all the registered methods of $C$ where $f_i$ has signature $f_i : e^i_1{:}T^i_1 \times ... \times e^i_{k_i}{:}T^{k_i}_1 \rightarrow T^i_0$. A proxy of $o$, $pxy_o$, is an object of type $PXY(C)$. The semantics of $pxy_o.f_i(p_0, ..., p_{k_i})$ is defined procedurally as follows:



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. **object import.** For all $j$, $1 \leq j \leq k_i$, where $e^i_j = in$, if $T^i_j$ is an object type $C'$, let $p'_j = LOID_{C'}(p_j)$; if $T^i_j = set < C' >$, where $C'$ is an object type, let $p'_j = \{LOID_{C'}(v) \mid \forall v \in p_j\}$. If $T^i_j$ is not an object type, let $p'_j = p_j$.

2. **method delegation.** $R' = o.f_i(p'_0, ..., p'_n)$.

3. **object export.** For all $j$, $1 \leq j \leq n$, where $e^i_j = out$, if $T^i_j$ is an object type $C'$, let $p_j = p'_j.PID()$, if $T^i_j = set < C' >$ where $C'$ is an object type, let $p_j = \{o.PID() \mid \forall o \in p'_j\}$. If $T^i_0$ is an object type $C'$, let $R = R'.PID()$. If $T^i_0 = set < C' >$ where $C'$ is an object type, let $R = \{o.PID() \mid \forall o \in R'\}$.

4. **return to client.** Return $R$.

□

The code that implements proxy types with interfaces as described in Definition 6.8.4, and semantics as described in Definition 6.8.5, must be generated by AURORA-OH mediators for all registered classes.

**Example 6.8.1** Consider the homogenizing view of $DS_2$ as shown in Table 6.1. The proxy type of *Pediatrician* should have the following interface:

```
class   PXY(Pediatrician)
{
        string          PID();
        string          TelNo();
        string          Profile();
        string          Specialization();
        short           YearsAdvancedTraining();
        setof(string)   PatientHistory( in string PatientID,
                        out string PreviousDoc, out date FirstAppt);
};
```

The only modification of interface from that of *Pediatrician* is in method *PatientHistory*; the proxy type replaces the type of the second parameter with **string**, the PID type of *Doctor*. □

### 6.8.4 Proxy Match Join: AURORA's Integration Operator

The goal of data integration in AURORA-OI is to use the proxies of the registered fragments of a global class $C_g$ to construct $C_g$ objects. These constructed objects, called *integrated objects*, are really *distributed objects* that *delegate* their methods for execution by appropriate proxies running at various AURORA-OH mediators. To the applications running at AURORA-OI, integrated objects appear to be usual objects that can be queried using OQL, and accessed using the programming language of choice. This section defines the integration operator used by AURORA-OI for manufacturing integrated objects. First, a few concepts must be presented.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

DEFINITION **6.8.6** [Intended Global Type.] Given a proxy $p$ representing an object of source type $C$ registered as a fragment of global class $C_g$, the *intended global type* of $p$, $IGT(p) = C_g$. □

Intended global type is the **semantic intension** of a proxy. This information is used to select proxies to delegate methods to.

DEFINITION **6.8.7** [Proxy Method Stub.] Let $M: e_1{:}T_1 \times ... \times e_n{:}T_n \rightarrow T_0$ be a method of global class $C_g$. Let $p$ be a proxy of type $PXY(C)$ where $C$ is a registered fragment of $C_g$. For a given object $o_g \in C_g$, $o_g.M$ is a *proxy method stub* of $p.M$ if the semantics of $R = o_g.M(p_0, ..., p_n)$. is equivalent to that described procedurally as follows:

1. **object export.** $\forall j,\ 1 \leq j \leq n$, where $e_j = in$, if $T_j$ is an object type, let $p'_j = p_j.PID()$; if $T_j = setof(C')$, where $C'$ is an object type, let $p'_j = \{o.PID() \mid \forall o \in p_j\}$. Otherwise, $p'_j = p_j$.
2. **method delegation to proxy.** $R' = p.M(p'_0, ..., p'_n)$.
3. **object import.** $\forall j,\ 1 \leq j \leq n$, where $e_j = out$, if $T_j$ is an object type $C'$, let $p_j = GOID_{C'}(p_j)$; if $T_j = setof(C')$, where $C'$ is an object type, let $p_j = \{GOID_{C'}(k) \mid \forall k \in p'_j\}$. If $R'$ is an object type $C'$, let $R = GOID_{C'}(R')$; if $R'$ is of type $setof(C')$, where $C'$ is an object type, let $R = \{GOID_{C'}(k) \mid \forall k \in R'\}$.
4. **Return result.** Return $R$.

□

Relating the above definition to Definition 6.8.5, one can see that proxy method stubs cooperate with proxies to facilitate exchange of oids between AURORA-OH and AURORA-OI mediators.

DEFINITION **6.8.8** [Proxy Match Join.] Let $C_g$ be a global class. Let $F_1, ..., F_m$ be all the registered fragments of $C_g$ or its descendent classes. Let $F_{m+1}, ..., F_{m+n}$ be all fragments registered with ancestor classes of $C_g$. Let $X_i (1 \leq i \leq m+n)$ be a collection of proxies such that $\forall o \in F_i. \exists p \in X_i$. $p$ is a proxy of $o$. The extent of $C_g$, $C_g.E$, is computed by operator *Proxy Match Join* (PMJ)

$$C_g.E = PMJ(C_g, m, X_1, ..., X_{m+n})$$

as defined below:

1. $C_g.E = \{o \mid \exists p \in X_1 \cup ... \cup X_m, o = GOID_{C_g}(p.PID())\}$. $\forall p \in X_1 \cup ... \cup X_{m+n}$, $p$ is a *contributing proxy* of object $o \in C_g.E$ if $p.PID() = GOID^{-1}_{C_g}(o)$. $CProxy(o)$ is used to denote all contributing proxies of object $o$.
2. $\forall o \in C_g.E$ and method $M$ of $C_g$, the semantics of $o.M$ should be equivalent to that of a proxy method stub of $p.M$ if $\exists p \in CProxy(o)$, such that $p$ supports $M$ and there exists no $p' \in CProxy(o)$, $p'$ supports $M$ but $IGT(p')$ is a descendent of $IGT(p)$. If no such $p$ exists, $o.M$ is a null method.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

□

The $PMJ$ operator constructs an object for each PID value appearing in fragments registered with $C_g$ or its descendents; these PIDs identify all objects in $C_g.E$. Contributing proxies may represent objects from fragments registered with $C_g$'s ancestor classes as well as those registered with $C_g$ and its descendent classes. When multiple contributing proxies are able to perform a given method, $PMJ$ selects the proxy with the most specific semantic intension; this is similar to the concept of late binding in object-oriented programming. AURORA-OI is responsible for generating an implementation for each method $M$ defined on $C_g$; such generated implementation must ensure that for any object $o \in C_g.E$, $o.M$ has this semantics specified in 2 of the above definition. Operator $PMJ$ is illustrated by the following example.

**Example 6.8.2** Consider class *Specialist* shown in Table 6.1 and assume the following:

$$Y_1 = \{p_{11}, p_{12}, p_{13}\},\ \ Y_{21} = \{p_{21}, p_{22}\},\ \ Y_{22} = \{p_{23}\},\ \ Y_3 = \{p_{31}, p_{32}\}$$

where $Y_1$, $Y_{21}$, $Y_{22}$ and $Y_3$ are collections of proxies of objects in fragments *MyDoctor* in $DS_1$, *Pediatrician* in $DS_2$, *FamilyPhysician* in $DS_2$, and *Orthopedics* in $DS_3$, respectively, $p_{ij}$ is a proxy of object $o_{ij}$ as shown in Table 6.1. Using the registration information in Table 6.1 and Definition 6.8.8:

$$specialists = PMJ(Specialist, 2, Y_{21}, Y_3, Y_1, Y_{21})$$

For illustration purpose, assume $GOID_{Specialist}(k)$ = *OI-Doctor-k*, where *Doctor* is the most general supertype of *Specialist*. For instance, the object in *"specialists"* with key 001 has oid *OI-Doctor-001*. By Definition 6.8.8, one gets:

| | |
|---|---|
| *specialists* | = {*OI-Doctor-001, OI-Doctor-004, OI-Doctor-002, OI-Doctor-007,* }, |
| *CProxy(OI-Doctor-001)* | = $\{p_{11}, p_{21}\}$, |
| *CProxy(OI-Doctor-004)* | = $\{p_{22}\}$, |
| *CProxy(OI-Doctor-002)* | = $\{p_{31}, p_{12}\}$, |
| *CProxy(OI-Doctor-007)* | = $\{p_{32}\}$, |

Behaviors of each object are derived using Definition 6.8.8. The details of *OI-Doctor-001* "Smith" are shown here. Two proxies, $p_{11}$ and $p_{21}$, contribute to Smith. The behaviors of Smith are:

| | |
|---|---|
| OI-Doctor-001.PID() | = "001"; |
| OI-Doctor-001.TelNo() | = $p_{21}$.TelNo(); |
| OI-Doctor-001.ClinicAddress() | = $p_{11}$.ClinicAddress(); |
| OI-Doctor-001.Profile | = $p_{21}$.Profile(); |

Both $p_{11}$ and $p_{21}$ can perform method *Profile*, but $p_{21}$ is chosen because it provides the profile of Smith as a *Specialist* - a more specific description than the profile of Smith as a generic *Doctor* $p_{11}$ provides. A related case is *OI-Doctor-002.Profile()*. 002 "Jones" is an orthopedics doctor but his profile as an orthopedics doctor is not available. However, Jones as a general doctor has a *Profile* and object



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

*OI-Doctor-002* will return this profile. This illustrates how objects registered with ancestor classes are used to construct integrated objects. The semantics of *OI-Doctor-001.PatientHistory(arg1, arg2, arg3)* should be the following according to Definition 6.8.7:

| | |
|---|---|
| returnValue = | $p_{21}$.PatientHistory(arg1, tempString, arg3); |
| arg2 = | $GOID_{Doctor}$(tempString); |
| Return | returnValue; |

□

## 6.9 Summary

This chapter described the object-oriented homogenization and integration framework of AURORA. The homogenization framework overlaps with object-oriented view frameworks, without operators for hiding properties from classes or hiding classes from inheritance hierarchies, and with an elaborate method mapping mechanism provided by operator *deriveOP* and the concept of navigation methods. The integration framework supports a simple object-matching assumption and manufactures global objects that perform their methods by dispatching them to appropriate source objects.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# Chapter 7

# Implementation of AURORA

Prototypes of AURORA's relational mediators (AURORA-RH and AURORA-RI) described in this dissertation have been implemented. The implementation is on Windows NT platform using the following tools: Microsoft Visual C++ 5.0 with Microsoft Foundation Classes (MFC), DB2/NT version 2.1.2, OLE-DB, and COM/DCOM.

## 7.1 Choosing a Distributed Computing Framework

The AURORA approach calls for three types of software components: the wrappers, the homogenization mediators, and the integration mediators. These components should cooperate to perform a mediation task. Implementation of AURORA is driven by two principles:

1. **Distribution.** To allow maximum flexibility, components should be allowed to run anytime, anywhere. AURORA components must be able to communicate with one another in the same way whether they reside on the same machine or not, and they should be able to activate another component when they need its services.

2. **Dynamic composition.** Since AURORA is built to facilitate large-scale data integration, the collection of AURORA components that makes up a data mediation system should expand/contract gracefully. Components should be allowed to join and leave the system freely; they should also be allowed to evolve without impacting on other components and the function of the system.

To support distribution, a distributed computing infrastructure is needed. To support dynamic composition, an agent is needed to locate components by name or identity, and to facilitate access to the components through pre-defined interfaces, so that the changes in a component do not impact on the use of it as long as the interfaces are maintained.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

An obvious choice for an infrastructure that satisfies all the above requirements is a distributed object computing (DOC) platform such as CORBA [70, 90] or COM/DCOM [61, 78]. These platforms allow transparent identification, activation, and accessing of objects locally or remotely; a new object supporting a known interface can join the mediation system simply by informing potential clients of its identity. These platforms also support access to objects through interfaces, which are contracts between objects and their clients; objects can evolve, but their clients will not have to modify their code or recompile; they are protected from such changes as long as the interfaces of the objects are maintained.

The choice of a DOC platform for AURORA implementation is not based on a detailed comparison of CORBA and COM. At a general level, it is clear that both CORBA and COM/DCOM would satisfy the requirements of AURORA implementation. Hence the choice was made based on availability of supporting technologies. COM/DCOM is chosen because in the COM/DCOM world, the advance in OLE-DB technology (Section 7.2.3) provides a strong commercial basis for wrappers. OLE-DB providers are COM components that provide uniform access to a variety of data sources. As shown in Section 7.3.2, with OLE-DB, the current implementation of AURORA does not need to build custom wrappers; commercial OLE-DB providers are available for a wide variety of data sources, and these providers can be composed to form wrappers. There is nothing similar to OLE-DB in the CORBA world. If CORBA is used as the base platform, custom wrappers for sources to be integrated must be built. Since wrapper technology is not a focus of AURORA research, building home-made wrappers would consume a significant amount of time without serving the purpose of demonstrating AURORA technology.

## 7.2 An overview of COM/DCOM and OLE-DB Technology

### 7.2.1 What is COM/DCOM?

The Component Object Model, COM, is a specification; it provides a standard that components and clients follow to ensure that they operate together. It specifies how to build components, also referred to as *servers*, that can be dynamically replaced without breaking the client code. In particular, it specifies what it means to be a COM component and how these components are accessed. The COM platform as provided by Microsoft consists of the following two closely related facilities:

1. The COM Library, an API that provides component management services that are useful for all clients and components. This library is written to guarantee that the most important operations are done in the same way for all components, and to save developers time in dealing with component management issues. Most of the COM library functions are built to provide support for distributed or networked components, rather than the local components.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. Distributed COM (DCOM) facilities on Windows systems provide the code needed to communicate with remote components. With DCOM, remote components can be accessed in exactly the same way as local servers. DCOM is not yet another model or specification for building components; it is the same as COM, but with a longer wire attached.

### 7.2.2 The COM Way of Building Systems

The COM provides a way to build a system as a collection of cooperative, possibly distributed, components. First, it is necessary to know what it means to be a COM component.

A COM component consists of executable code distributed as Win32 dynamic linking libraries (DLLs) or executables (EXEs). COM components must be written to meet all the requirements prescribed by the COM specification. Programming details aside, on a higher level, these components must satisfy the following requirements:

1. Dynamic linking. Components must be able to link at run time. This ensures that components can evolve or be replaced without breaking the client code.
2. Encapsulation. Clients must be protected from the implementation details of the components; components must maintain stable interface(s). This means that the components must satisfy the following conditions:
   (a) Clients should be able to use any components regardless of the programming languages used to write the client or the component.
   (b) Components must be shipped in binary form, compiled, linked, ready to use.
   (c) Components must be upgradable without breaking client code.
   (d) Components must be transparently relocatable on a network; remote components should be treated in exactly the same way as the local ones.

The COM library provides a variety of support for building components that satisfy the above requirements. Numerous hooks must be built into the component programs. These hooks are easy to build once the programmer understands how they work. Given below are high level descriptions of various aspects of COM components and their relationship with the clients.

**Identification and Activation of COM components**

COM components are executables identified by *Class Identifiers* (CLSIDs), which are *Globally Unique Identifiers* (GUIDs). A GUID is a 128-bit structure that is programmatically generated, based on the computer on which it was created and the time at which it is generated. GUIDs are globally unique, although they are generated without coordination with any central authority. COM components are registered with the Window's registry and can be launched by COM API



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

functions. To gain access to a component, all a client needs is the CLSID of the components; COM library functions and/or DCOM support on Windows are responsible for setting up the connection and facilitating exchange of data. Construction and access of remote components can be done in exactly the same way as local ones; the DCOM support on Windows makes the component location transparent to the clients.

**Accessing COM components**

COM components support one or more interfaces. When a client connects to a component through a COM API function, it must identify the interface to be used. If the connection is successful, then the client gets back a pointer to the desired interface. The client can then use this interface pointer to access the services of the component and to get access to any other interface the component supports. Interfaces are identified by *Interface identifiers* (IIDs), which are also GUIDs. The client wishing to use a particular interface must know its IID.

**Interfaces**

To a client, a component is a set of interfaces. The client can only access a component through an interface. The client has little knowledge of a component as a whole, other than the interfaces that provide the services of interest. A client is often not completely aware of all the interfaces a component supports. Interfaces are described using the *Interface Definition Language* (IDL), a C++-like language with extensions, and without implementation of any methods defined. Compiling of an IDL file will produce code that provides both the client and the components with necessary facilities to use or support all the interfaces described, including the IID of all the interfaces of interest. Note that interfaces are not tied to any component: one interface might be implemented by many components and one component can implement many interfaces. Interfaces can inherit from other interfaces, but this inheritance does not imply any relationship between the components that implement these interfaces. From the surface, this seems to stop code reuse, which is an important feature of OO programming. However, COM allows code reuse through *containment* and *aggregation*. These features will not be further discussed, since AURORA implementation does not require them.

### 7.2.3 OLE-DB Technology

OLE-DB is a specification of the standard interfaces of a specific type of COM components, the ones that provide access to a wide variety of data sources. Such interfaces range from those used for connecting to a source, starting and closing a session, retrieving schema information, sending queries, to those dealing with data as a set of rows. An *OLE-DB provider* is an implemented COM component that supports some of the interfaces specified by the OLE-DB specification. OLE-DB



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

specification states that certain interfaces are "mandatory", that is, all OLD-DB providers must support them. Often an OLE-DB provider allows access to a specific type of data source and supports a subset of the interfaces specified by OLE-DB specification, depending on the capabilities of the underlying data source. Example sources with OLE-DB providers are sources with an ODBC driver, email archives, and spreadsheets. The advent of the OLE-DB technology is notable for two impacts on the software industry:

1. It allows a broader range of data sources to be accessed through a standard interface. In the past, the Open Database Connectivity (ODBC) was the omnipresent methodology for providing access to data sources, but it typically only provides access to sources with database capabilities.

2. It opens up data sources for access through COM/DCOM. An OLE-DB provider is just another COM component and it can be accessed with all the convenience provided by the COM/DCOM platform.

With the fast growing popularity of OLE-DB technology, an increasing number of data sources are accessible through their OLE-DB providers. The rest of this section describes the OLE-DB technology in more detail.

An OLE-DB provider is a COM component and hence is activated when a client requests it using the CLSID of the provider. Upon activation, the provider first creates a *data source object* specified by OLE-DB as below:

```
TDataSource {
    [mandatory] interface IDBCreateSession;
    [mandatory] interface IDBInitialize;
    [mandatory] interface IDBProperties;
    [mandatory] interface IPersist;
    [optional]  interface IConnectionPointContainer;
    [optional]  interface IDBAsynchStatus;
    [optional]  interface IDBDataSourceAdmin;
    [optional]  interface IDBInfo;
    [optional]  interface IPersistFile;
    [optional]  interface ISupportErrorInfo;
}
```

A data source object must support all the mandatory interfaces and may support some of the optional ones. To get access to data, the client uses the `IDBcreateSession` interface to gain access to a *session object*, which supports the following interfaces:

```
TSession {
    [mandatory] interface IGetDataSource;
    [mandatory] interface IOpenRowset;
    [mandatory] interface ISessionProperties;
    [optional]  interface IDBCreateCommand;
```



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    [optional]  interface IDBSchemaRowset;
    [optional]  interface IIndexDefinition;
    [optional]  interface ISupportErrorInfo;
    [optional]  interface ITableDefinition;
    [optional]  interface ITransaction;
    [optional]  interface ITransactionJoin;
    [optional]  interface ITransactionLocal;
    [optional]  interface ITransactionObject;
}
```

Depending on whether the data source is able to accept commands, the session object may or may not support the IDBcreateCommand interface. If the underlying data source is an ODBC data source, then this interface is supported and the client can use it for sending queries and getting the result back. If a data source is not able to entertain queries, this interface is often not supported. In this case, the client uses the IOpenRowset interface to access data in tabular form. Various scenarios of using an OLE-DB provider for data access are depicted in Figure 7.1.

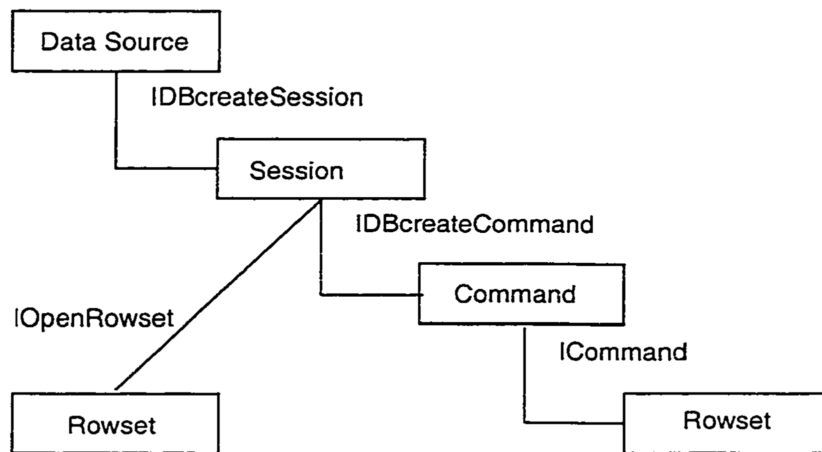



Figure 7.1: Use of OLE-DB Interfaces and the Role of Rowsets

As shown above, all OLE-DB providers must be able to provide data in tabular form, as *rowsets*. Indeed, rowsets are the central objects that enable all OLE-DB providers to expose data in tabular form. Conceptually, a rowset is a set of rows in which each row has columns of data. Base table providers present their data in the form of rowsets. Query processors present the result of queries in the form of rowsets. Even schema information is provided as rowsets. The interfaces of rowsets, as specified by OLE-DB, are given below:

```
TRowset {
    [mandatory] interface IAccessor;
    [mandatory] interface IColumnsInfo;
```



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    [mandatory] interface IConvertType;
    [mandatory] interface IRowset;
    [mandatory] interface IRowsetInfo;
    [optional]  interface IChapteredRowset;
    [optional]  interface IColumnsRowset;
    [optional]  interface IConnectionPointContainer;
    [optional]  interface IDBAsynchStatus;
    [optional]  interface IRowsetChange;
    [optional]  interface IRowsetFind;
    [optional]  interface IRowsetIdentity;
    [optional]  interface IRowsetLocate;
    [optional]  interface IRowsetResynch;
    [optional]  interface IRowsetScroll;
    [optional]  interface IRowsetUpdate;
    [optional]  interface IRowsetView;
    [optional]  interface ISupportErrorInfo;
}
```

The most basic rowset object exposes five interfaces: `IRowset`, which contains methods for fetching rows in the rowset sequentially; `IAccessor`, which permits the definition of groups of column bindings describing the way tabular data is bound to consumer program variables; `IColumnsInfo`, which provides information about the columns of the rowset; and `IRowsetInfo`, which provides information about the rowset itself. Using IRowset, a consumer can sequentially traverse the rows in the rowset, including traversing backward if the rowset supports it. The interface `IRowset` includes the following methods: `AddRefRows`, that adds a reference count to an existing row handle; `GetData` that retrieves data from the rowset's copy of the row; `GetNextRows` that fetches rows sequentially, remembering the previous position; `ReleaseRows` that releases rows; `RestartPosition` that repositions the next fetch position to its initial position - that is, its position when the rowset was first created.

## 7.3 AURORA Mediators as COM Components

AURORA components are implemented as COM components that cooperate across networked computers, as shown in Figure 7.2. These components should support pre-defined interfaces.

### 7.3.1 Interfaces of AURORA Components

AURORA wrappers and mediators support pre-defined interfaces which describe the services offered. Generally, all AURORA mediators support the following services:

1. Schema export service: this service should allow the schema supported by the mediator to be accessed. Depending on the data model of the mediator, this schema can be relational or object-oriented.

2. Query service: this service should accept queries posed against the schema of the mediator.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Depending on the data model of the mediator, the queries can be posed in SQL or OQL. Object-oriented mediators should also support access via an object-oriented database programming language.

3. Event notification service: this service notifies the clients of events of interest.

In the current implementation, interfaces of components are custom and are supported only by demand of demonstrating the base technology of AURORA. In the future, these services should be defined to standard, such as the OMG object services, and be fully supported. Currently supported interfaces are described in Sections 7.4.1 and 7.4.2.

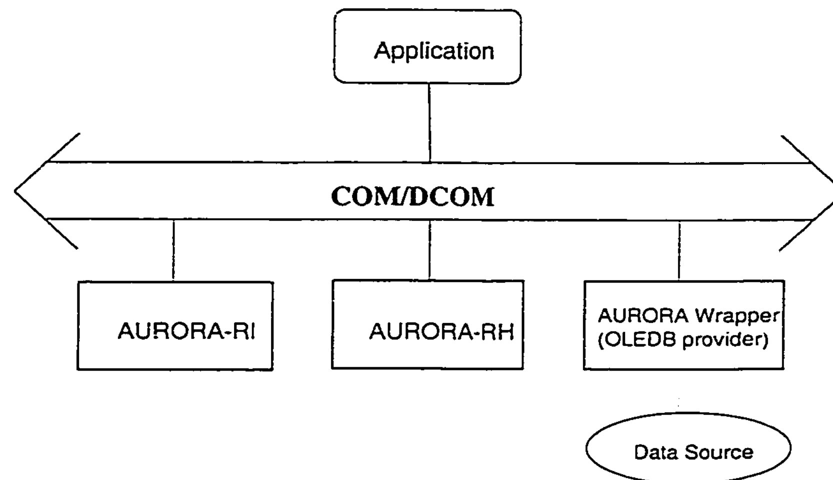



Figure 7.2: AURORA Components as COM Components

### 7.3.2 OLE-DB Providers as AURORA Wrappers

Microsoft provides an OLE-DB provider for all sources with an ODBC driver. This OLD-DB provider supports interfaces for connecting to a database, retrieving schema information, sending SQL queries, and collecting query results as rowsets. This set of functions is sufficient for uniform access of sources with SQL capabilities. For data sources without SQL query capabilities, there are commercial middleware products (e.g., the ISG Navigator, which are OLE-DB providers themselves) that add SQL query capabilities to any OLE-DB provider that does not support it. Such middleware can be used as an "adaptor" that transforms non-SQL OLE-DB providers into an SQL provider.

In the current implementation, AURORA wrappers are OLE-DB providers supporting SQL queries. As such, both OLE-DB providers for ODBC sources and OLE-DB providers for a middleware such as the ISG Navigator can be used as wrappers. This wrapper strategy is illustrated in



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

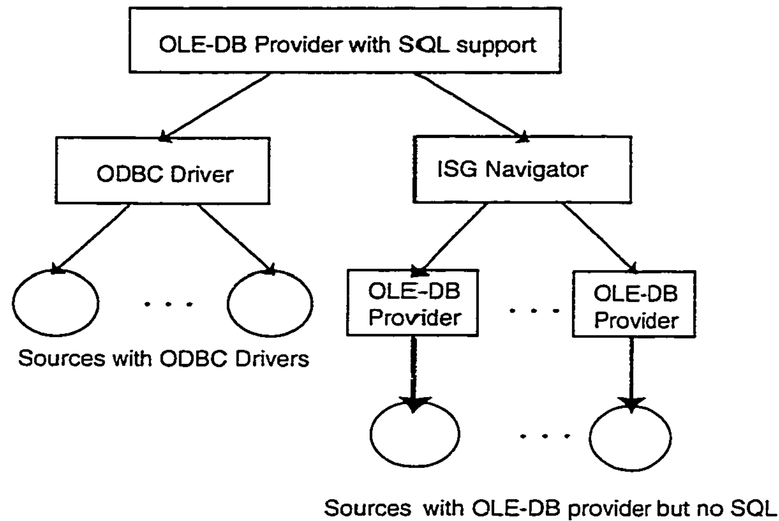



Figure 7.3: The Making of AURORA Wrappers Using OLE-DB Providers

Figure 7.3. A directed line from A to B in this figure means "A can access B". While these wrappers do not support services to any standard, as described earlier, it will be easy to build a generic layer on top of these wrappers to support interfaces to desired standard. More significantly, by employing this wrapper strategy, the current implementation of AURORA is able to access a variety of data sources without building custom wrappers. The focus of the implementation work is on the design and implementation of AURORA-RH and AURORA-RI mediators.

## 7.4 Implementation of AURORA-RH and AURORA-RI

AURORA's relational mediators, AURORA-RH and AURORA-RI, have been implemented to form a framework for dynamic integration of data sources with OLE-DB providers. The canonical data model supported by this implementation is the relational data model. These mediators are implemented as components that cooperate through the COM/DCOM framework. It is necessary to look at AURORA-RH and AURORA-RI together to show what they do, and why, and how they work together. The current AURORA implementation is illustrated in Figure 7.4.

### 7.4.1 Implementation of AURORA-RH

As shown in Figure 7.4, implementation of AURORA-RH consists of two parts: (1) implementation of MAT-RH; and (2) implementation of the AURORA-RH query server.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

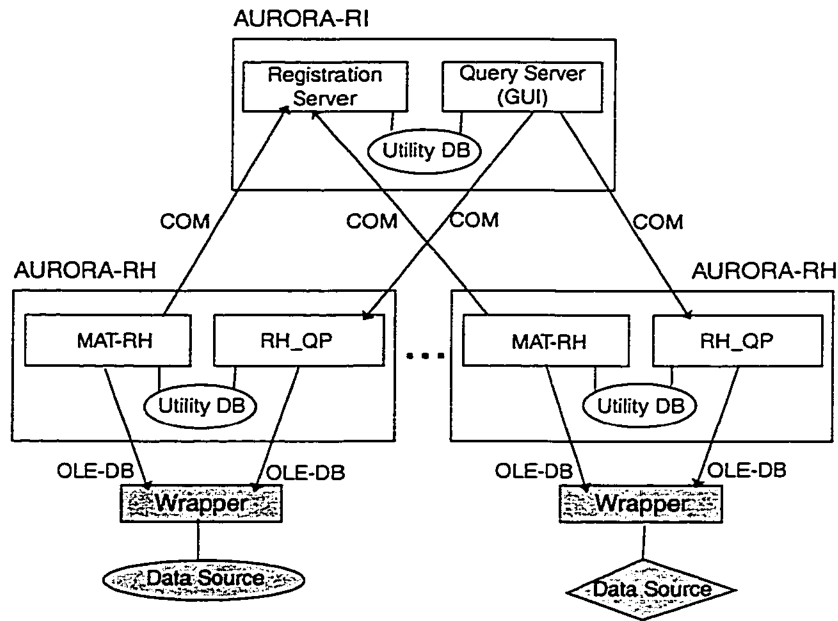



Figure 7.4: Implementation of AURORA-R Mediators

### MAT-RH

MAT-RH is built as a graphical user interface that allows mediator authors to connect to a data source, browse its source schema, create a homogenizing view by specifying virtual relations and their derivations, and finally, register some or all relations in the homogenizing view with an AURORA-RI mediator. As shown in Figure 7.5, the top level menu consists of the following items:

1. `Initialize`. Selecting this item would activate a pop-up menu that allows the user to perform the following tasks:

    - Choosing the item `Initialize` allows initializing the AURORA-RH mediator to a named data source. Upon initialization, the system retrieves the schema of the data source and allows the user to browse it. Internally, the system also creates a companion utility database with a schema designed for storing mapping information. This database is later used for storing derivations of the homogenizing view and for manipulating temporary tables during query processing.
    - `Save To DB`. Choosing this item would cause the mappings that are specified to be saved into the companion utility database. This is usually performed after the homogenizing view has been constructed completely.

2. `Import`. Choosing this item will activate a pop-up menu that provides two options for importing part or all of the source schema: `Import whole schema` and `Import by query`. Another



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

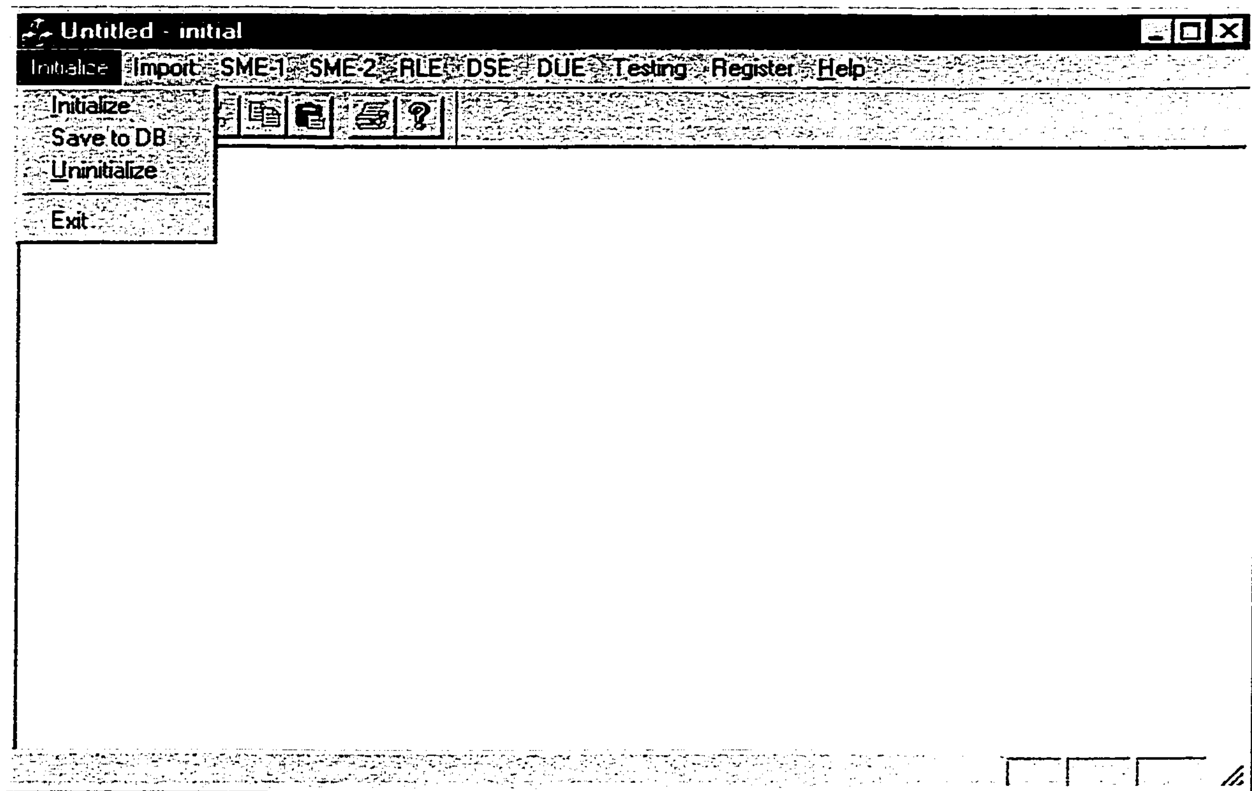



Figure 7.5: Main Window of MAT-RH GUI

item on the pop-up menu allows the imported schema to be browsed. This pop-up menu is shown in Figure 7.6.

3. `SME-1`. Choosing this item will bring up the pop-up menu that provides various facilities for resolving type 1 cross-over mismatch. As shown in Figure 7.7, the items on this pop-up menu include the following:

   - `ViewDef with Pad`. This item allows the use of the *pad* primitive for defining derived relations. A dialog window will pop up to collect various parameters for a *pad* operation.
   - `ViewDef with Rename`. This item allows renaming of relations.
   - `ViewDef with RELmat`. This item allows the use of the *RELmat* transformation. Choosing this item will bring up a dialog window, as shown in Figure 7.8. The user fills up the entries in this dialog and a view relation will be derived by MAT-RH.
   - `ViewDef with Query`. This item allows the user to specify a view relation as a relational query over all the relations derived so far.
   - `Display View Schema`. This item allows the user to browse the schema that includes all the view relations derived so far.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

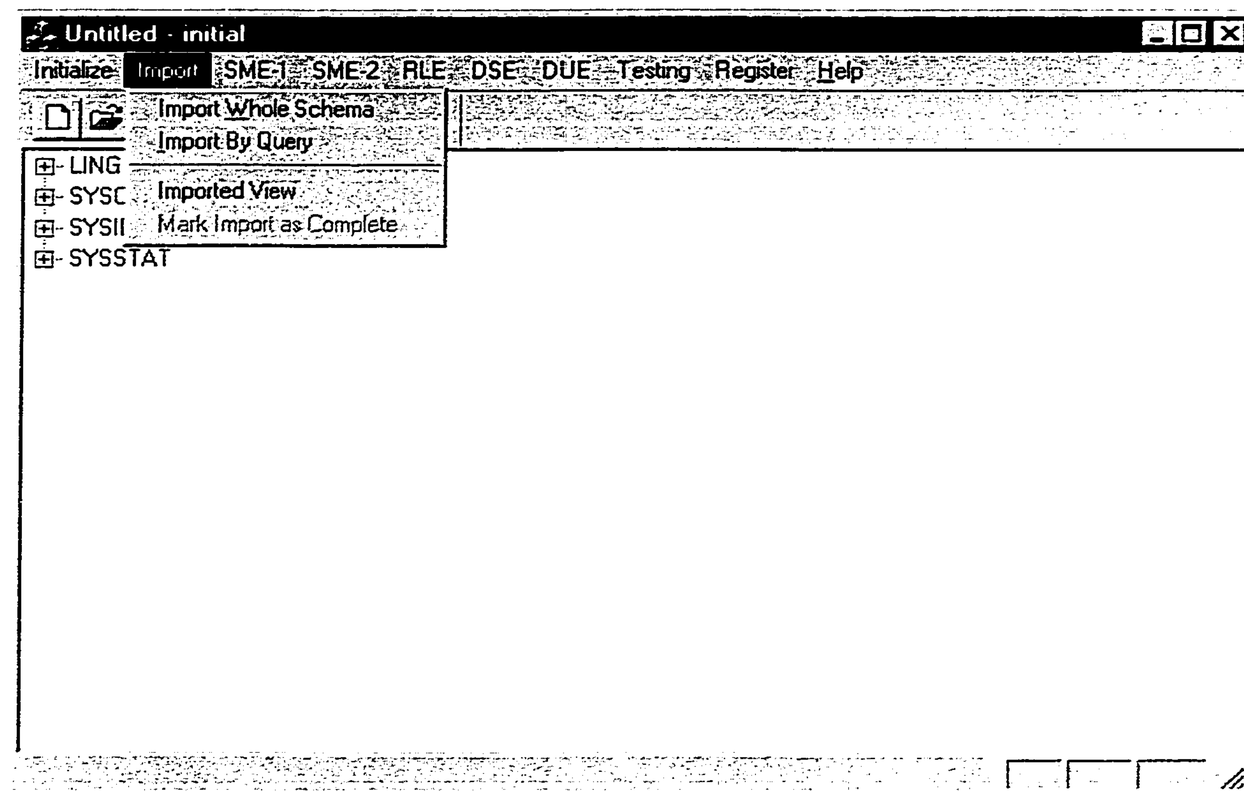



Figure 7.6: Import pop-up menu

4. `SME-2`. Choosing this item will start the pop-up menu that provides various facilities for resolving type 2 cross-over mismatch. The items on this pop-up menu are similar to those on SME-1 except that `ViewDef with RELmat` is replaced by `ViewDef with ATTRmat`. The dialog window brought up by choosing `ViewDef with ATTRmat` is shown in Figure 7.9.

5. `RLE`. Choosing this item will start the pop-up menu that provides various facilities for relation linking. As shown in Figure 7.10, this pop-up menu includes the following items:

   - `ViewDef with Query`. This item allows derivation of view relations using relational queries over all the relations derived so far.
   - `Select Prototype Relations`. This item allows the user to mark existing relations as the prototype relations.
   - `Display View Schema`. After prototype relations have been specified. the view schema will include only these relations and are often smaller than before.

6. `DSE`. Choosing this item will bring up the pop-up menu that provides facilities for resolving domain structural mismatches by specifying domain structural functions (DSFs). The pop-up menu includes two items:



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

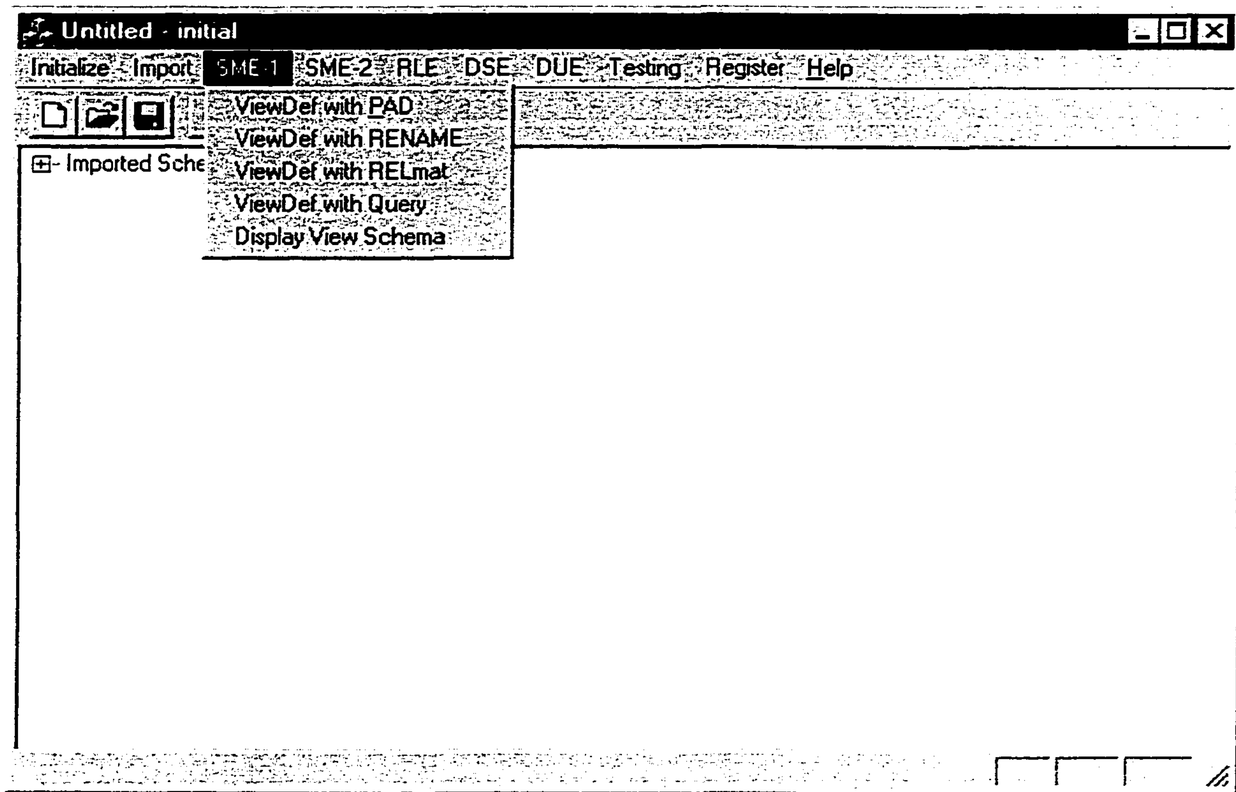



Figure 7.7: SME-1 pop-up menu

- `Specify Domain Structural Functions`. Choosing this item will bring up a sequence of dialogs, shown in Figures 7.11 and 7.12, that allows the mediator author to specify the list of attributes in a prototype relation that correspond to an attribute in the target relation, and to declare a domain structural function of the appropriate signature. The DSFs themselves must be provided by the user as a DLL. In the current implementation, all such DLLs must be in a single file. In the future, this restriction will be removed.
- `Display View Schema`. Displays the current schema.

7. `DUE`. Choosing this item will bring up the pop-up menu that provides facilities for resolving domain element mismatches by specifying domain value functions (DSFs). The facilities provided are similar to those in DSE, except the functions declared in this environment are domain value functions, which are functions that take one in-parameter and return a value. Inverses of these functions, if they exist, are also accepted by the system for later use.

8. `Register`. Choosing this item will bring up a sequence of dialogs, shown in Figures 7.13 and 7.14, which allow the user to register one source relation as a fragment of a target relation. Ideally, the registration process should allow the mediator author to specify the name of the target AURORA-RI mediator, and the target relation of which a source relation is a



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

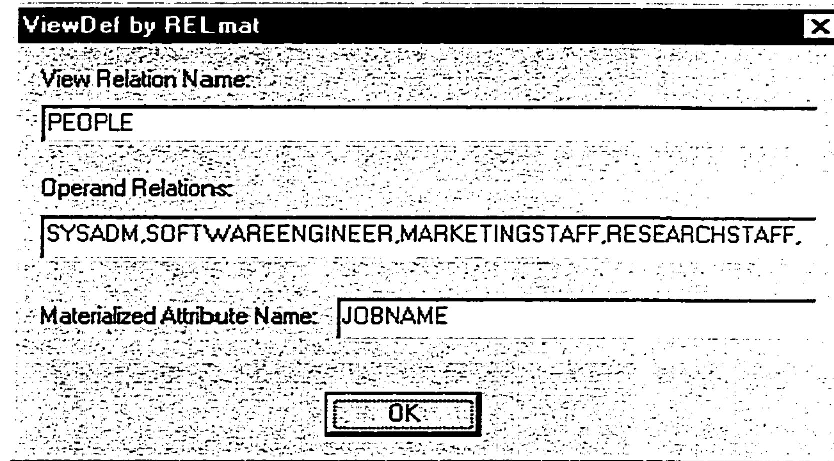



Figure 7.8: Dialog Window for the *RELmat* Transformation

fragment. In the current implementation, there is only one AURORA-RI mediator running and its identity is hard-coded in the AURORA-RH mediators; hence the user does not choose which AURORA-RI mediator to register data with. This restriction will be removed in the future.

Usually a mediator author works with MAT-RH as follows. First, she connects to the data source of interest. Then she invokes the 6 tools in sequence to remove various types of mismatches by deriving view relations. This process is guided by the homogenization methodology. Once she is satisfied with the homogenizing view derived, she must do two things before exiting MAT-RH. First, she must save the homogenizing view and its derivation to the utility database. This is done by choosing `Initialize → Save to DB`. Second, she must register some or all of the relations in the derived view as fragments of relations in the target service view, supported by the AURORA-RI mediator. This is done by using the registration facilities described earlier.

### AURORA-RH Query Server: RH_QP

AURORA-RH query server is a COM server. It supports a single interface, `IRHQuery`, as shown in the IDL specification below, `IRHQuery` consists of two methods: `ExecQuery` and `GetNextRow`. `ExecQuery` accepts queries given as three strings: the select clause, the from clause, and the where clause. `GetNextRow` returns a row of data in a buffer. This interface is used by AURORA-RI mediators to send queries for execution and to retrieve query results.

```
//
```



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

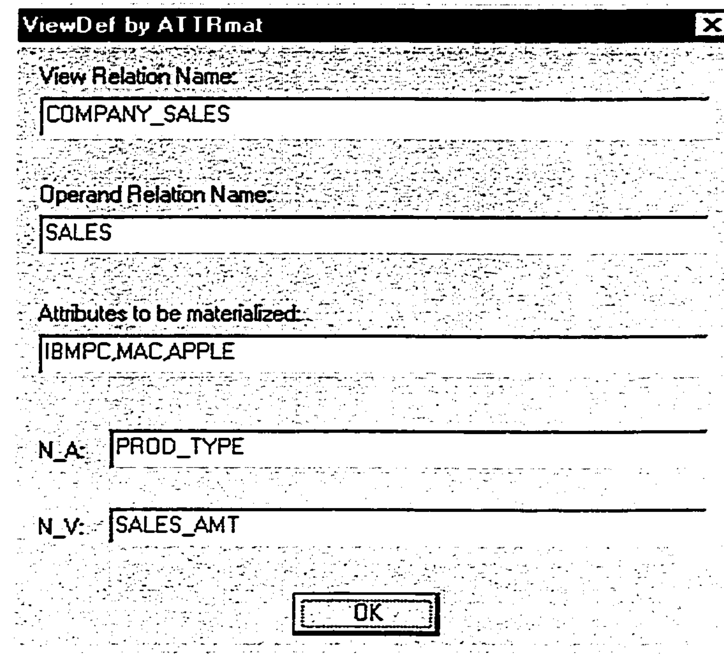



Figure 7.9: Dialog Window for the *ATTRmat* Transformation

```
// RH_QP.idl
//
import "unknwn.idl" ;
//
// Interface IRHQuery
//
[
    object,
    uuid(05EDAE72-D6EA-11d1-A811-0004AC9592CC),
    helpstring("Query Interface of AURORA-RH"),
    pointer_default(unique)
]
interface IRHQuery : IUnknown
{
    HRESULT ExecQuery ( [in, string] wchar_t* DataSourceName,
                        [in, string] wchar_t* selectClause,
                        [in, string] wchar_t* fromClause,
                        [in, string] wchar_t* whereClause
                      );
    HRESULT GetNextRow ( [out] int* succ,
                         [out] wchar_t RowBuffer[1000]
                       );
};
//
// Component descriptions
//
```



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

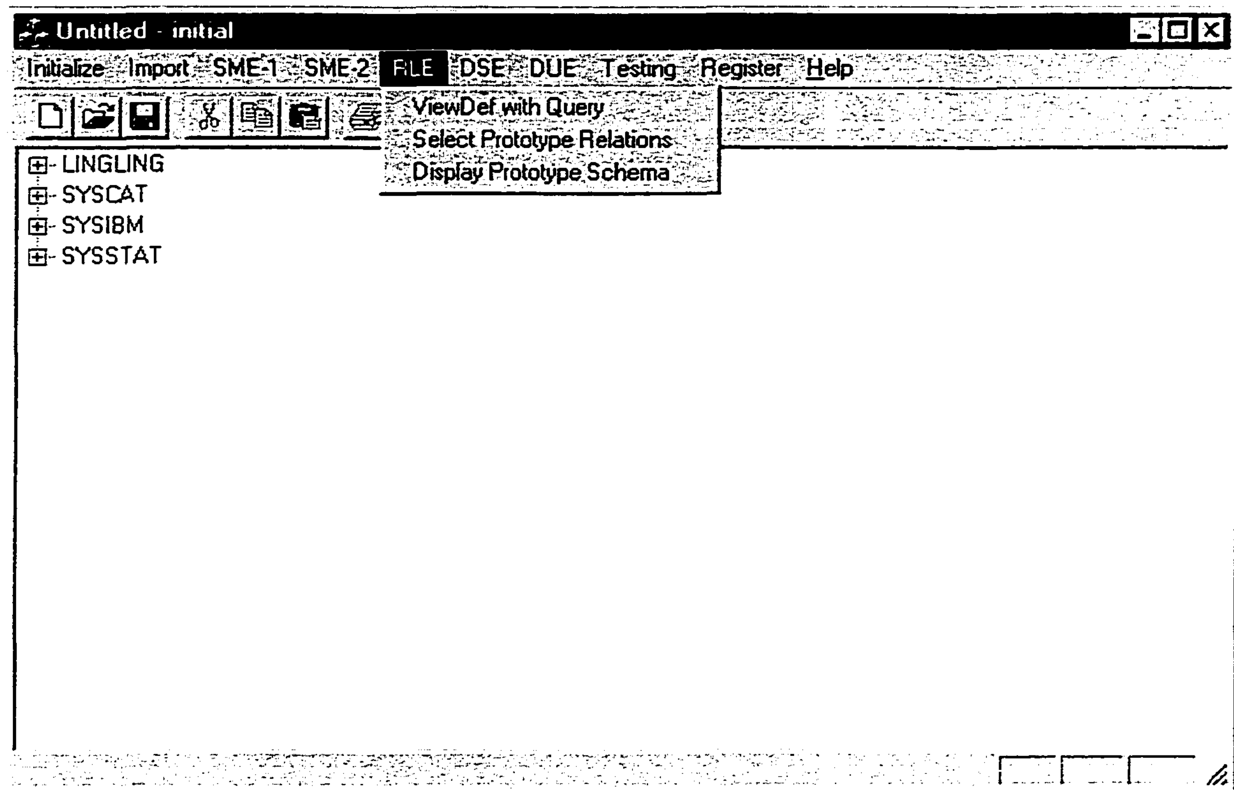



Figure 7.10: Pop-up Menu of RLE

```
[
    uuid(887365AD-D479-11d1-A811-0004AC9592CC),
    version(1.0),
    helpstring("RHQP 1.0 Type Library")
]
library RHQP_Lib
{
    importlib("stdole32.tlb") ;
    [
        uuid(05EDAE74-D6EA-11d1-A811-0004AC9592CC),
        helpstring("AURORA-RH Query Processor Component")
    ]
    coclass RHQP_CMPNT
    {
        [default] interface IRHQuery ;
    };
} ;
```

Upon receiving a query against a data source $D$, RH_QP first loads the homogenizing view of source $D$ from the companion utility database of $D$. It then rewrites the query using view mappings loaded from the utility database, to generate an initial QEP. This QEP is then transformed into a more efficient QEP. Currently, RH_QP is able to employ all the transformation rules shown in Table 5.5 that involve selection to optimize the initial QEP. Once the optimized QEP is generated, RH_QP



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

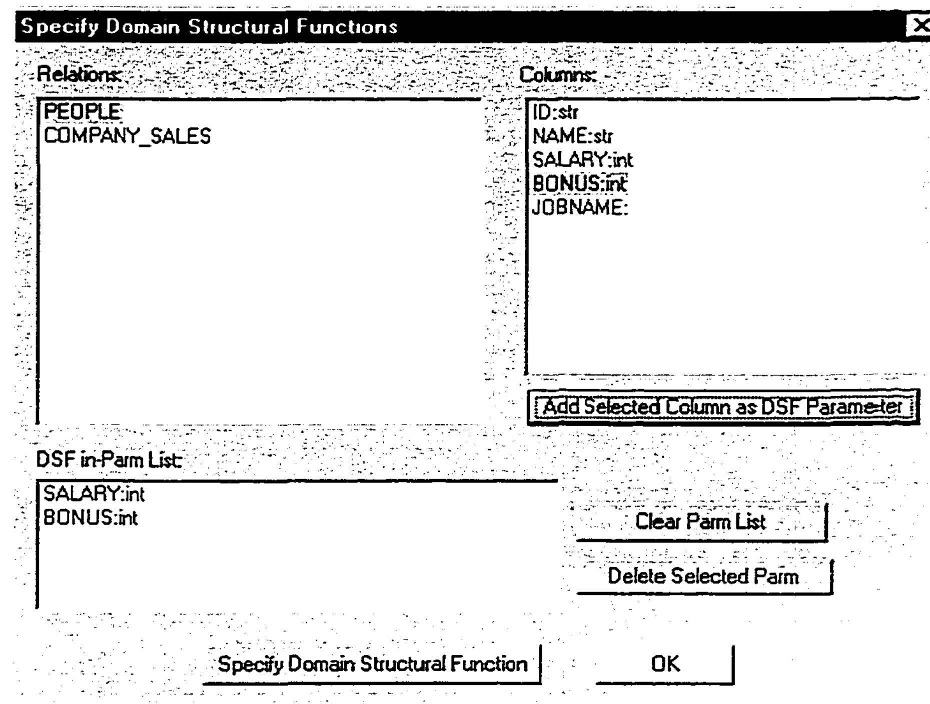



Figure 7.11: First Dialog for Specifying Domain Structural Functions

will evaluate the QEP bottom up. First, it sends subqueries to source $D$, then it assembles the returned results to produce the final query result. Query results can be retrieved by the client using the `GetNextRow` method in the `IRHQuery` interface supported by RH_QP..

### 7.4.2 Implementation of AURORA-RI

AURORA-RI implementation includes two parts: the registration server, and the query server. The registration server is a COM component supporting a single interface, `IRIRegister`, which includes a single method, `RegisterFragment`. This method is invoked by MAT-RH for registering source relations. It accepts five parameters: `hostName`, the name of the host that the AURORA-RH mediator is running at; `sourceDBName`, the name of the source database; `sourceRelName`, the name of the source relation to be registered; `targetRelName`, the name of the global relation in the service view of which the relation named by `sourceRelName` is a fragment; and `fragmentScheme`, the scheme of the relation named by `sourceRelName`.

```
//
// RIREG.idl
//
import "unknwn.idl" ;
typedef struct
{
```



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

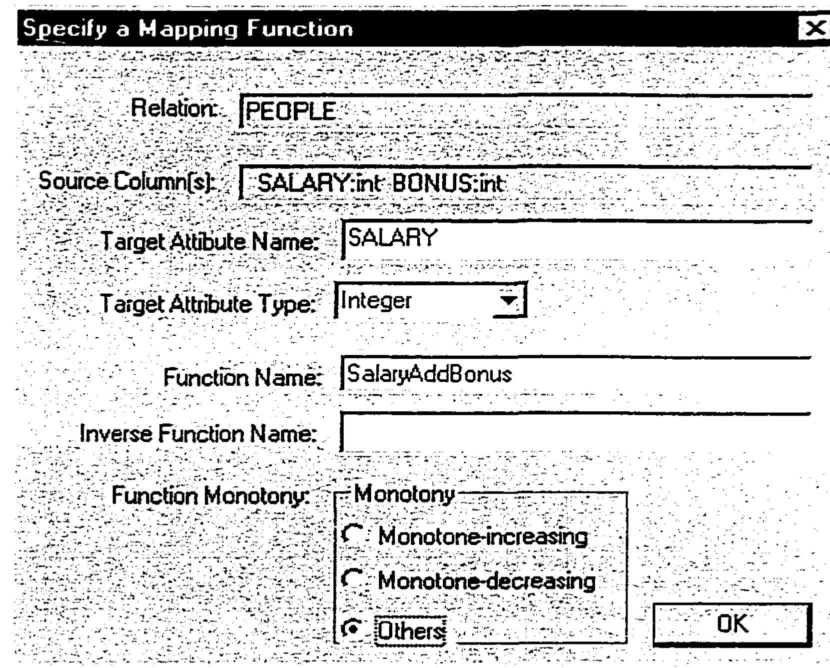



Figure 7.12: Second Dialog for Specifying Domain Structural Functions

```
    int         numColumns;
    wchar_t     columnName[50][60];
    int         colType[50];
} XRelScheme;
//
// Interface IRIRegister
//
[
    object,
    uuid(9E900F31-DBA0-11d1-A81B-0004AC9592CC), // Apr 24 2:15pm
    helpstring("Registration Interface of AURORA-RI"),
    pointer_default(unique)
]
interface IRIRegister : IUnknown
{
    HRESULT RegisterFragment (  [in, string] wchar_t*      hostName,
                                [in, string] wchar_t*      sourceDBName,
                                [in, string] wchar_t*      sourceRelName,
                                [in, string] wchar_t*      targetRelname,
                                [in]         XRelScheme    fragmentScheme
                             );
};
//
// Component descriptions
//
[
```



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

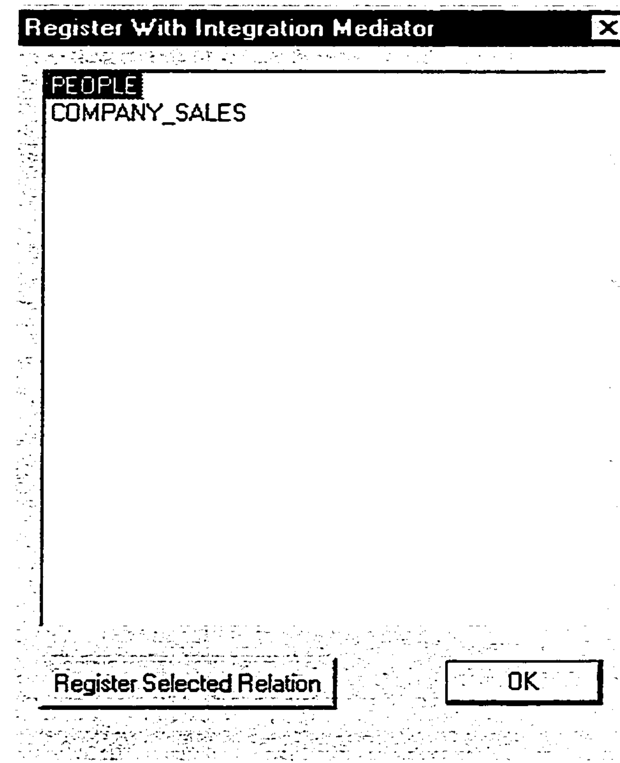



Figure 7.13: First Dialog for Registering Relations with Integration Mediator

```
    uuid(9E900F33-DBA0-11d1-A81B-0004AC9592CC),
    version(1.0),
    helpstring("RIREG 1.0 Type Library")
]
library RIREG_Lib
{
    importlib("stdole32.tlb") ;
    [
        uuid(9E900F35-DBA0-11d1-A81B-0004AC9592CC),
        helpstring("AURORA-RI Registration Server")
    ]
    coclass RIREG_CMPNT
    {
        [default] interface IRIRegister ;
    };
} ;
```

Upon receiving a registration, the AURORA-RI registration server will store the registration information in a utility database. The content of this database will then be used by the AURORA-RI query server to decide where and how to collect fragments when needed.

The AURORA-RI query server is currently a GUI driven program. The user launches the query server and is presented with a GUI with which the supported service view can be browsed and queries



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

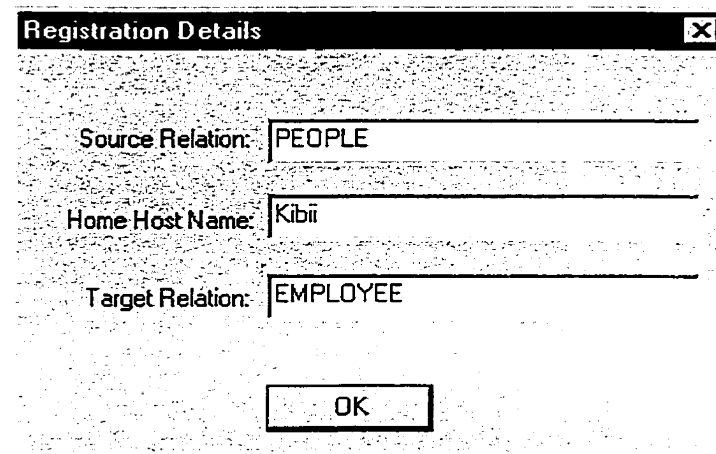



Figure 7.14: Second Dialog for Registering Relations with Integration Mediator

can be posed. Upon receiving a query, the query server goes to the utility database to find out which fragments are needed for query processing. Currently, there are no query optimization capabilities implemented, so the query server always retrieves all the relevant fragments in full, performs a match join to produce the relation(s), and then processes the query.

## 7.5 Observations and Experiences

Using a DOC framework is an elegant way of supporting distribution and dynamic composing of AURORA components. Work done in order to ship the components as COM servers is limited: the COM hooks in the client and server programs are mostly reusable. Generally, the amount of work involved to use the COM platform is small in comparison to the work required for implementing the logistics of the components.

Both AURORA-RH and AURORA-RI have a query engine. These engines do not implement their own join, selection, buffer management etc., but rather use these facilities provided by a commercial RDBMS, DB2/NT. This strategy works, but it requires creating temporary tables to hold intermediate results during mediator query processing, which can be quite slow. It is probably a much better idea to expand an existing query processor with operators in AURORA. This requires access to and knowledge of a good, existing, query processor.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# Chapter 8

# Conclusions and Future Work

This dissertation describes the AURORA project. Research in AURORA established new techniques for building an integrated data access middleware system (IDAMS) based on the mediator architecture. A prototype system is built to demonstrate the paradigm and techniques developed. This chapter summarizes the contributions and experiences of AURORA, and reviews future work.

## 8.1 Contributions

Research in AURORA has two main dimensions: (1) a 2-tiered mediation model, flexible data model support, and mediation methodologies; and (2) Mediator query processing based on Mediation Enabling Algebras (MEAs) and the conflict tolerant query model. The first dimension deals with the general paradigm of a data mediation system, the second dimension focuses on developing enabling techniques.

### 8.1.1 Mediation Model, Flexible Data Models, and Mediation Methodologies

A mediation model prescribes various tasks in the data integration process (also referred to as data mediation) and the relationship among them. AURORA's 2-tiered mediation model prescribes that data mediation be performed in two steps: homogenization followed by integration. This mediation model enables a divide-and-conquer approach towards integration of a large number of heterogeneous and autonomous data sources. It enables scalable mediation, where adding and removing data sources is easy. AURORA supports both relational and object-oriented data models: mediators are available in both models. Consequently, AURORA mediators are *function specific*, performing specific mediation tasks; and *data model specific*, supporting either the relational or the object-oriented (ODMG) data model. This paradigm enables scalable integration of a wide range of



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

data sources:

1. Wrapper construction work is reduced due to the flexible data model support in AURORA. Data sources can be wrapped with a relational or an object-oriented interface, whichever is most easily generated. As demonstrated in AURORA, relational wrappers can be composed using commercial middleware systems. In general, the AURORA paradigm does not require data sources to "upgrade" their data models and thus allows them to participate in the integration scope without incurring major wrapper construction work.

2. The 2-tiered mediation model of AURORA divides the data integration task into pieces that can be worked on independently and in parallel. The difficulties encountered in building large scale data integration systems originate from two sources: semantics and scale. Working with semantics is difficult, working with semantic differences among a large number of heterogeneous sources is even more difficult. A general principle in building AURORA is "semi-automatic homogenization, automatic integration". Homogenization deals with a wide range of semantic issues but concerns single sources, while integration deals with a small number of semantic issues although the number of sources involved is large. Furthermore, homogenization mediators in AURORA are equipped with a mediator author's toolkit (MAT) that helps the mediator authors to work with semantics. The basis of a MAT is a mediation methodology that guides a mediator author to work with semantic issues systematically. MATs also provide various facilities to the mediator author. A MAT has been built as part of the AURORA-RH mediator in the prototype system.

### 8.1.2 Scalability and Flexibility of the AURORA Approach

Scalability of the AURORA approach is enabled through the 2-tiered mediation model, which controls the complexity of building a large-scale data integration system by prescribing a divide-and-conquer approach. Adding and removing of data sources are easy. The complexity of the data integration activity does not increase with the number of sources involved; large-scale integration can be performed as easily as integration of a small number of sources.

Flexibility of the approach is enabled also by the 2-tiered mediation model and by the flexible data model support provided with all the AURORA mediators. The 2-tiered mediation model prescribes that adding and removing of data sources do not impact on the availability and validity of the service view, or the participation of other data sources. Unavailable data sources can be treated as a source that removed itself from the access scope voluntarily. Flexible data model support of AURORA enables the applications and the data sources to consume/contribute data based on a data model that is most comfortable; this greatly increases the practical appeal of a data integration system.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

### 8.1.3 Enabling Techniques

Enabling techniques are mainly in the area of processing mediator queries. Two types of techniques have been proposed: query processing and optimization based on Mediation Enabling Algebras (MEAs), and Conflict Tolerant (CT) query models.

Data manipulations that are specific to mediation systems, and are unknown to traditional DBMSs, are captured in MEAs which are the basis for algebraic and cost-based mediator query optimization and processing. Different mediators employ different MEAs. MEAs for all AURORA mediators have been defined. In the relational mediators, query processing techniques based on MEAs have also been established. With MEAs, the impact of the mediation process on query processing has been identified and taken into consideration during query processing.

The Conflict Tolerant (CT) query models are employed by AURORA integration mediators for querying multi-source data. The CT query models represent a step away from the traditional paradigm of querying data integrated from multiple sources. Rather than creating a single-source illusion, conflicts are exposed to the applications in a controlled and manageable manner. The key point in designing a CT query model is to provide enough levels of conflict tolerance to cater for most application requirements, without overwhelming the applications with complicated choices in conflict handling. This approach allows the applications more control over how conflicts are handled and provides the mediation systems with more space for query optimization, especially when conflict rate is low. Currently, only the CT query model employed by the AURORA-RI mediator is completed.

## 8.2 Experiences

The AURORA experience gives rise to two observations on research in IDAMSs. First, building IDAMSs involves issues in both paradigms and techniques; adopting a new paradigm gives rise to new technical problems. From a pragmatic view point, employing a paradigm that applications can identify with is equally important as developing techniques to make the system function efficiently. AURORA chose to employ a new paradigm and study the related technical problems. In terms of research, a drawback is that the validity and applicability of the techniques developed depend on that of the general paradigm employed; it is more difficult to demonstrate their significance.

Second, choice of paradigms varies with target applications. The choice of the paradigm in AURORA was not made based on an in-depth study of several classes of applications, but rather based on one specific type of application - the electronic commerce application - and the potential for producing technical results. In hindsight, a more thorough study of application scenarios may provide more input into the design of the paradigm.

The current implementation of AURORA realizes the vision of using light-weight, specialized, easy-to-use components to build increasingly sophisticated data mediation systems. With the ad-



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

vances in Internet and distributed computing technology, monolithic mediation systems with a static access scope will no longer be sufficient. Future mediation systems must be highly distributed and must be able to expand/reduce their access scopes gracefully. The current prototype system demonstrates that the AURORA approach allows such mediation systems to be constructed.

Two observations can be made after the implementation of the AURORA prototype. First, making use of currently available software can reduce the workload tremendously. AURORA makes use of commercial middleware systems, such as OLD-DB provides and the ISG Navigator, to form wrappers, and hence avoids building custom wrappers while gaining access to a wide range of data sources. Second, the efficiency of data exchange and manipulation of mediators requires more work. Currently, the AURORA prototype system focuses on demonstrating the paradigm and the base techniques; data exchange among, and data manipulation within, the mediators are inefficient. This is an engineering issue and may require a detailed study of the state-of-the-art technology available.

## 8.3 Future Work

Future work on extending AURORA itself falls into the following categories:

1. Extending the flexible data model support in AURORA to include XML. This extension will eventually allow Web pages to be treated as data sources. XML is regarded as the main medium that allows Web data manipulation and exchange. Currently, considerable work is going on in querying and managing XML data sources. Progress in this regard may provide inspirations for building AURORA mediators based on XML.

2. Further establishing of the conflict tolerant querying facilities. The CT query model for object-oriented integration mediator is not yet defined. Definition of the CT query model for relational mediators requires extension. In particular, the CT query semantics may become unclear in complicated queries such as nested queries, or queries involving aggregation functions. The number of levels of tolerance can be reviewed in the future. The key is to offer adequate support for the applications to deal with conflicts at run-time, without overwhelming the applications with complicated handling of these conflicts.

3. To carry on with the idea of exposing instance level conflicts to the applications, rather than hiding them at high system expenses, the CT query paradigm may be adapted to deal with other types of conflicts, such as lineage and source credibility of data items. The challenge is to design enough number of tolerance modes without leaving the applications overwhelmed.

Work that further establishes the AURORA approach on a more formal basis also gives rise to interesting future research topics, as discussed below.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**Formal Semantics of the AURORA Approach.**

As described in Section 3.2.3, the formal semantics of queries posed against a service view in AURORA requires more work. Moreover, the introduction of CT queries may give rise to new issues in completeness and soundness of a global database, as well as query semantics. Such work can be carried out in a fashion similar to that of [33] and should provide insight into the value of systems such as AURORA.

**Criteria for Evaluating IDAMSs.**

Choosing an IDAMS that work well can be a complex evaluation process since it involves issues such as transformation and composition of application semantics, usability of the facilities, etc. Some part of this evaluation may not have a formal basis. For now, a few criteria may be useful, including the following:

1. Completeness of the range of mismatches and conflicts a system is capable of handling. The question to be answered is: *"Given any data integration scenario, is the system capable of performing all required data conversion, matching and combination for integration purpose?"*

2. Providence of mediation methodologies and their completeness. Most previous systems do not provide a mediation methodology while AURORA does. However, AURORA's mediation methodologies do not have a formal basis and their completeness is yet to be established. The ultimate question to be answered is the following: *"Given any data integration scenario, do the mediation methodologies provided enable the users to identify all differences and overlaps among the data sources, and, resolve them correctly? Do the mediation methodology guarantee a correct data integration?"*.

3. Ease of use of the system, whether the users are provided with enough facilities to perform manual data integration tasks.

4. Efficiency of data manipulation within the integration system, whether redundant data retrieval is minimized.

5. Safety of queries, whether the queries posed against the integrated (virtual) data have a well-defined, deterministic semantics.

The first two criteria of the evaluation lead to a more fundamental issue: how do we formally describe, compare, transform, and, merge multiple application models to produce a new application model, required by a class of applications? To do this, a formal model is required for describing the semantics of the source data as well as the semantics of the desirable target data. Once such formal models are established, one can formally identify a complete range of mismatches that must be handled by a good IDAMS. One can also formally establish the completeness of a mediation



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

methodology, by proving that it mandates the removal of all possible mismatches and guarantees a correct data integration.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# Bibliography


[1] S. Abiteboul and A. Bonner. Objects and Views. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 238–247, 1991.

[2] ISG Navigator Universal Data Access. *www.isgsoft.com/products/Navigator/*. ISG International Software Group.

[3] S. Adali, K.S. Candan, Y. Papakonstantinou, and V.S. Subrahmanian. Query Caching And Optimization In Distributed Mediator Systems. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 137–148, 1996.

[4] R. Ahmed and R. Rafii. Relational Schema Mapping and Query Translation in Pegasus. In *Proceedings of the Workshop on Multidatabase and Semantic Interoperability*, pages 22–25, 1990.

[5] R. Ahmed, P. Smedt, W. Du, W. Kent, M. Ketabchi, W. Litwin, A. Rafii, and M. Shan. The Pegasus Heterogeneous Multidatabase System. *IEEE Computer*, 24(12):19–27, December 1991.

[6] Y. Arens, C.Y. Chee, C-N. Hsu, and C.A. Knoblock. Retrieving and Integrating Data From Multiple Information Sources. *International Journal of Intelligent and Cooperative Informations Systems*, pages 127–158, June 1993.

[7] C. Batini, M. Lenzerini, and S. Navathe. A Comparative Analysis of Methodologies of Database Schema Integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[8] R. Bayardo, B. Bohrer, R. Brice, A. Cichocki, J. Fowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. Semantic Integration of Information in Open and Dynamic Environments. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 195–206, 1997.

[9] J.A. Blakeley, W.J. McKenna, and G. Graefe. Experiences Building the Open OODB Query Optimizer. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 287–296, 1993.

[10] D. Brill, M. Templeton, and C. Yu. Distributed Query Processing Strategies in Mermaid, A Frontend to Data Management Systems. In *Proceedings of the First International Conference on Data Engineering*, pages 211–218, 1984.

[11] P. Buneman, S.B. Davidson, G.G. Hillebrand, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 505–516, 1996.

[12] M. Carey, L. Haas, P. Schwarz, M. Arya, W. Cody, R. Fagin, M. Flickner, A. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. Williams, and E. Wimmer. Towards Heterogeneous Multimedia Information Systems: the Garlic Approach. In *Fifth Int. Workshop on Research Issues in Data Engineering – Distributed Object Management (RIDE-DOM'95)*, pages 124–131, 1995.

[13] M. Castellanos and F. Saltor. Semantic Enrichment of Database Schema: An Object-Oriented Approach. In Y. Kambayashi, M. Rusinkiewicz, and A. Sheth, editors, *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems (IMS 91)*, 1991.




Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[14] R. Cattell and D. Barry. *The Object Databse Standard: ODMG 2.0.* Morgan Kaufmann. 1997.

[15] A. Chen. Outerjoin Optimization in Multidatabase Systems. In *Proceedings of the 2nd International Symposium on Distributed and Parallel Database Systems*, pages 211–218, 1990.

[16] C. Chung. Dataplex: An Access to Heterogenous Distributed Databases. *Communications of the ACM (CACM)*, 33(1):70–80, January 1990.

[17] C. Collet, M. Huhns, and W. Shen. Resource Integration Using a Large Knowledge Base in Carnot. *IEEE Computer*, 24(12):55–62, December 1991.

[18] B. Czejdno, M. Rusinkiewicz, and D. Embley. An Approach to Schema Integration and Query Formulation in Federated Database systems. In *Proceedings of the Third International Conference on Data Engineering*, pages 477–484, 1987.

[19] K.H. Davis and A.K. Arora. Converting a Relational Database Model into an Entity-Relationship Model. In S.T March, editor, *Entity-Relationship Approach*, pages 551–572. 1988.

[20] U. Dayal. Processing queries over generalization hierarchies in a multidatabase system. In *Proceedings of 9th International Conference on Very Large Data Bases*, page 342=353, 1983.

[21] U. Dayal. Query Processing in a Multidatabase System. In W. Kim, D. Reiner, and D. Batory, editors, *Query Processing in Database Systems*, pages 81–108. Springer-Verlag, 1985.

[22] U. Dayal and H-Y. Hwang. View Definition and Generalization for Database Integration in a Multidatabase System. *IEEE Transactions on Software Engineering*, SE-10(6):628–645, November 1984.

[23] L. Demichiel. Performing Operations Over Mismatched Domains. In *Proceedings of the Fifth International Conference on Data Engineering*, pages 36–45, 1989.

[24] A. Dogac, C. Dengi, and M.T. Ozsu. Building Interoperable Databases on Distributed Object Management Platforms. *Communication of ACM (CACM)*, 41(9):95–103, September 1998.

[25] W. Du, R. Krishnamurthy, and M. Shan. Query Optimization in a Heterogeneous DBMS. In *Proceedings of 18th International Conference on Very Large Data Bases*, pages 277–291, 1992.

[26] H. Duchene, M. Kaul, and V. Turau. VODAK Kernel Data Model. In Klaus R. Dittrich, editor, *Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems*, pages 242–261, 1988.

[27] C. Evrendilek, A. Dogac, S. Nural, and F. Ozcan. Multidatabase Query Optimization. *Distributed and Parallel Databases*, 5:77–114, 1997.

[28] D. Florescu, L. Raschid, and P. Valduriez. Using Heterogeneous Equivalences for Query Rewriting in Multidatabase Systems. In *Proceedings of the Third International Conference on Cooperative Information Systems (CoopIS-95)*, pages 158–169, 1995.

[29] D. Florescu, L. Raschid, and P. Valduriez. Defining the Search Space For Query Optimization In A Heterogeneous Database Management System. In *Under Review*, 1996.

[30] G. Gardarin, B. Finance, and P. Fankhauser. Federating Object-Oriented and Relational Databases: The IRO-DB Experience. In *Proceedings of the Second IFCIS International Conference on Cooperative Information Systems (CoopIS 97)*, pages 2–13, 1997.

[31] C H. Goh, M E. Madnick, and M D. Siegel. Ontologies, Context, And Mediation: Representing And Reasoning About Semantic Conflicts In Heterogeneous And Autonomous Systems. Working Paper 3848, MIT Sloan School of Management, 1995.

[32] D. Goldhirsh and L. Yedwab. Processing Read-Only Queries Over Views With Generalization. In *Proceedings of 10th International Conference on Very Large Data Bases*, pages 344–348, 1984.

[33] Gsta Grahne and Alberto O. Mendelzon. Tableau Techniques for Querying Information Sources through Global Schemas. In Catriel Beeri and Peter Buneman, editors, *Database Theory - ICDT '99, 7th International Conference, Proceedings, Lecture Notes in Computer Science, Vol. 1540*, pages 332–347. Springer, 1999.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[34] J. Grant, W. Litwin, N. Roussopoulos, and T. Sellis. An Algebra and Calculus for Relational Multidatabase Systems. In Y. Kambayashi, M. Rusinkiewicz, and A. Sheth, editors, *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems (IMS 91)*, 1991.

[35] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing Queries Across Diverse Data Sources. In *Proceedings of 23rd International Conference on Very Large Data Bases*, pages 276–285, 1997.

[36] J. Hammer and D. McLeod. An Approach to Resolving Semantic Heterogeneity in a Federation of Autonomous, Heterogeneous Database Systems. *International Journal of Intelligent and Cooperative Information Systems*, 2(1):51–83, 1993.

[37] U. Hohenstein and C. Korner. A Graphical Tool for Specifying Semantic Enrichment of Relational Databases. In *IFIP TC-2 Working Conference on Data Semantics (DS-6)*. 1995.

[38] H. Hwang, U. Dayal, and M. Gouda. Using Semiouterjoins to Process Queries in Multidatabase Systems . In *ACM PODS*, pages 153–162, 1984.

[39] L.A. Kalinichenko. Methods and Tools for Equivalent Data Model Mapping Construction. In *Advances in Database Technology - EDBT'90. International Conference on Extending Database Technology*, pages 92–119, 1990.

[40] M. Kaul, K. Drosten, and E. Neuhold. ViewSystem: Integrating Heterogeneous Information Bases by Object-Oriented Views. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 2–10, 1990.

[41] W. Kent. Solving Domain Mismatch and Schema Mismatch Problems with an Object-Oriented Database Programming Language. In *Proceedings of 17th International Conference on Very Large Data Bases*, pages 147–160, 1991.

[42] W. Kim, I. Choi, S. Gala, and M. Scheevel. On Resolving Schematic Heterogeneity In Multidatabase Systems. *Distributed and Parallel Databases*, 1(3):251–279, 1993.

[43] W. Kim and W. Kelley. On View Support in Object-Oriented Database Systems. In Won Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, pages 108–129. Addison Wesley, 1995.

[44] W. Kim and J. Seo. Classifying Schematic and Data Heterogeneity in Multidatabase Systems. *IEEE Computer*, 24(12):12–18, December 1991.

[45] R. Krishnamurthy, W. Litwin, and W. Kent. Language Features For Interoperability Of Databases With Schematic Discrepancies. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 40–49, 1991.

[46] V.S. Lakshmanan, F. Sadri, and I.N. Subramanian. SchemaSQL: A Language for Interoperability in Relational Multi-database Systems. In *Proceedings of 22nd International Conference on Very Large Data Bases*, pages 239–250, 1996.

[47] T. Landers and R. Rosenberg. An overview of Multibase. In H J Schneider, editor, *Distributed Databases*, pages 153–184. North-Holland, Netherland, 1982.

[48] A. Levy. Obtaining Complete Answers from Incomplete Databases. In *Proceedings of 22nd International Conference on Very Large Data Bases*, pages 402–412, 1996.

[49] A. Levy, A. Rajaraman, and J. Ordille. Query Answering Algorithms for Information Agents. In *Proceedings of the 13th National Conference on Artificial Intelligence, AAAI-96,*, pages 40–47, 1996.

[50] A. Levy, A. Rajaraman, and J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proceedings of 22nd International Conference on Very Large Data Bases*, pages 251–262, 1996.

[51] W. Litwin. MSQL: A multidatabase Language. *Information Sciences*, 49(1-3):59–101, October 1990.

[52] W. Litwin and Ph. Vigier. Dynamic Attributes in the Multidatabase System MRDSM. In *Proceedings of the Second International Conference on Data Engineering*, pages 103–110, 1986.




Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[53] L. Liu, C. Pu, and Y. Lee. An Adaptive Approach To Query Mediation Across Heterogeneous Information Sources. In *Int. Conf. on Cooperative Information Systems (CoopIS)*, pages 144–156, June 1996.

[54] Ling Liu. Query Routing in Large-Scale Digital Library Systems. In *Proceedings of the Fifteenth International Conference on Data Engineering*, pages 154–163, 1999.

[55] G. Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternative. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 18–27, 1988.

[56] H J. Lu, B C. Ooi, and C H. Goh. On Global Multidatabase Query Optimization. *SIGMOD Record*, 21(4):6–11, December 1992.

[57] V.M. Markowitz and J.A. Markowsky. Identifying Extended Entity-Relationship Object Structures in Relational Schemas. *IEEE Trans. on Software Engineering*, 16(8), August 1990.

[58] W. Meng and C. Yu. Query Processing in Multidatabase Systems. In Won Kim, editor, *Modern Database Systems: The Object Model, Interoperability and Beyond*, pages 551–572. Addison-Wesley Publishing Company, 1995.

[59] W. Meng, C. Yu, and W. Kim. Processing Hierarchical Queries in Heterogeneous Environments. In *Proceedings of the Eighth International Conference on Data Engineering*, pages 394–401, 1992.

[60] W. Meng, C. Yu, W. Kim, G. Wang, T. Pham, and S. Dao. Construction Of Relational Front-end For Object-Oriented Database Systems. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 476–483, 1993.

[61] Microsoft. The Component Object Model Specification. *http://www.microsoft.com/oledev/olecom/title.htm*, 1995.

[62] R. Miller. Using Schematically Heterogeneous Structures. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 189–200, 1998.

[63] R. Miller, Y. Ioannidis, and R. Ramakrishnan. Schema Equivalence in Heterogeneous Systems: Bridging Theory and Practice. *Information Systems*, 19(1):3–31, 1994.

[64] R. Miller, Y.E. Ioannidis, and R. Ramakrishnan. The Use of Information Capacity in Schema Integration and Translation. In *Proceedings of 19th International Conference on Very Large Data Bases*, pages 120–133, 1993.

[65] P. Missier. Extending A Multidatabase Language To Resolve Schema and Data Conflicts. Master's thesis, University of Houston, 1993.

[66] P. Missier and Mark Rusinkiewicz. Extending a Multidatabase Manipulation Language To Resolve Schema And Data Conflicts. In *IFIP TC-2 Working Conference on Data Semantics (DS-6)*, pages 93–115, 1995.

[67] A. Motro. Superviews: Virtual Integration Of Multiple Databases. *IEEE Trans. on Software Engineering*, SE-13(7):785–798, July 1987.

[68] A. Motro. Multiplex: A formal model for multidatabases and its implementation. *Technical Report ISSE-TR-95-103, Department of Information and Software System Engineering, George Mason University*, 1995.

[69] OMG. *The Common Object Request Broker Architecture and Specification (CORBA)*. Object Management Group, 1992.

[70] OMG. *The Common Object Request Broker Architecture and Specification (CORBA). 2.0*. Object Management Group, March 1995.

[71] F. Ozcan, S. Nural, P. Koksal, C. Evrendilek, and A. Dogac. Dynamic Query Optimization on a Distributed Object Management Platform. In *Proceedings of Fifth International Conference on Information and Knowledge Management (CIKM)*, pages 117–124, 1996.

[72] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object Fusion in Mediator Systems. In *Proceedings of 22nd International Conference on Very Large Data Bases*, pages 413–424, 1996.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[73] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A Mediation System Based on Declarative Specifications. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 132–141, 1996.

[74] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260, 1995.

[75] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A Query Translation Scheme for Rapid Implementation of Wrappers. In *International Conference on Deductive and Object-Oriented Databases*, pages 161–186, 1995.

[76] X. Qian. Query Folding. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 48–55, 1996.

[77] X. Qian and T F. Lunt. Semantic Interoperation: A Query Mediation Approach. Technical Report SRI-CSL-94-02, Computer Science Laboratory, SRI International, April 1994.

[78] D. Rogerson. *Inside COM.* Microsoft Press, 1997.

[79] M. Roth, F. Ozcan, and L. Haas. Cost Models Do Matter: Providing Cost Information for Diverse Data Sources in a Federated System. In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 599–610, 1999.

[80] M. Roth and P. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *Proceedings of 23rd International Conference on Very Large Data Bases*, pages 266–275, 1997.

[81] E.A. Rundensteiner. Multiview: a Methodology for Supporting Multiple View Schemata in Object-Oriented Databases. In *Proceedings of 18th International Conference on Very Large Data Bases*, pages 187–198, 1992.

[82] M. Rusinkiewicz. OMNIBASE: Design and Implementation of a Multidatabase System. In *Proceedings of the 1st Annual Symposium in Parallel and Distributed Processing*, pages 162–169, 1989.

[83] S.Agarwal, A. Keller, G. Wiederhold, and K. Saraswat. Flexible Relation: an Approach for Integrating Data From Multiple, Possibly Inconsistent Databases. In *Proc. 11th Int'l. Conf. on Data Engineering*, pages 495–504, 1995.

[84] E. Sciore, M. Siegel, and A. Rosenthal. Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems. *ACM Transactions on Database Systems (TODS)*, 19(2):254–290, June 1994.

[85] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[86] L. Suardi, M. Rusinkiewicz, and W. Litwin. Execution of Extended Multidatabase SQL. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 641–650, 1993.

[87] V S. Subrahmanian, S. Adali, A. Brink, R. Emery, J. Lu, A. Rajput, T. Rogers, R. Ross, and C. Ward. HERMES: Heterogeneous Reasoning And Mediator System. Unpublished document, University of Maryland.

[88] M. Templeton, H. Henley, E. Maros, and D.J. Van Buer. InterViso: Dealing With the Complexity of Federated Database Access. *VLDB Journal*, 4(2):287–317, 1995.

[89] A. Tomasic, L. Raschid, and P. Valduriez. Scaling Heterogeneous Databases And The Design Of Disco. In *Proceedings of the International Conference on Distributed Computer Systems*, pages 449–457, 1996.

[90] S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 14(2), February 1997.

[91] Gio Wiederhold. Mediators In The Architecture Of Future Information Systems. *IEEE Computer*, pages 38–49, March 1992.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[92] E. Wong and R. Katz. Logical Design and Schema Conversion for Relational and DBTG Databases. In P.P Chen, editor, *Entity-Relationship Approach to System Analysis and Design*. 1980.

[93] L. Yan and T. Ling. Translating Relational Schema with Constraints into OODB Schema. In *IFIP DS-5 Semantics of Interoperable Database Systems*, 1992.

[94] L L. Yan. Towards Efficient and Scalable Mediation: the AURORA Approach. In *Proceedings of the IBM CASCON Confernece*, pages 15–29, 1997.

[95] L L. Yan and T. Ozsu. Conflict Tolerant Queries in AURORA. In *Proc. 4th IFCIS Conference on Cooperative Information Systems (CoopIS-99)*, pages 279–290, 1999.

[96] L L. Yan, T. Ozsu, and L. Liu. Towards a Mediator Development Environment: The AURORA Approach. Technical Report TR-96-21, Department of Computing Science, University of Alberta, August 1996.

[97] L L. Yan, T. Ozsu, and L. Liu. Accessing Heterogeneous Data Through Homogenization and Integration Mediators. In *Proc. 2nd IFCIS Conference on Cooperative Information Systems (CoopIS-97)*, pages 130–139, 1997.

[98] L L. Yan, T. Ozsu, and L. Liu. Mediator Join Indices. In *Seventh International Workshop on Research Issues in Data Engineering: High-Performance Database Management for Large Scale Applications (RIDE'97)*, pages 51–59, 1997.

[99] C. Yu, C. Chang, M. Templeton, D. Brill, and E. Lund. Query Processing in a Fragmented Relational Distributed System. *IEEE Transaction on Software Engineering*, 11(8):795–810, 1985.

[100] C. Yu, L. Lilien, K. Guh, M. Templeton, D. Brill, and A.L.P. Chen. Adaptive Techniques for Distributed Query Optimization. In *Proceedings of the Second International Conference on Data Engineering*, pages 86–93, 1986.

[101] C. Yu, Y. Zhang, W. Meng, W. Kim, G. Wang, T. Pham, and S. Dao. Translation of Object-Oriented Queries to Relational Queries. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 90–97, 1995.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.