Linköping Studies in Science and Technology
Dissertation No. 582

Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration

Vanja Josifovski

Department of Computer and Information Science
Linköpings universitet
S-581 83 Linköping, Sweden
Linköping 1999

Linköping Studies in Science and Technology
Dissertation No. 582

Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration

Vanja Josifovski

INSTITUTE OF TECHNOLOGY
LINKÖPINGS UNIVERSITET

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 1999

Abstract

An important factor of the strength of a modern enterprise is its capability to effectively store and process information. As a legacy of the mainframe computing trend in recent decades, large enterprises often have many isolated data repositories used only within portions of the organization. The methodology used in the development of such systems, also known as *legacy systems*, is tailored according to the application, without concern for the rest of the organization. From organizational reasons, such isolated systems still emerge within different portions of the enterprises. While these systems improve the efficiency of the individual enterprise units, their inability to interoperate and provide the user with a unified information picture of the whole enterprise is a "speed bump" in taking the corporate structures to the next level of efficiency.
Several technical obstacles arise in the design and implementation of a system for integration of such data repositories (sources), most notably distribution, autonomy, and data heterogeneity. This thesis presents a data integration system based on the wrapper-mediator approach. In particular, it describes the facilities for passive data mediation in the AMOS*II* system. These facilities consist of: (i) object-oriented (OO) database views for reconciliation of data and schema heterogeneities among the sources, and (ii) a multidatabase query processing engine for processing and executing of queries over data in several data sources with different processing capabilities. Some of the major data integration features of AMOS*II* are:

- A distributed mediator architecture where query plans are generated using a distributed compilation in several communicating mediator and wrapper servers.
- Data integration by reconciled OO views spanning over multiple mediators and specified through declarative OO queries. These views are

capacity augmenting views, i.e. locally stored attributes can be associated with them.

- Processing and optimization of queries to the reconciled views using OO concepts such as overloading, late binding, and type-aware query rewrites.
- Query optimization strategies for efficient processing of queries over a combination of locally stored and reconciled data from external data sources.

The AMOS*II* system is implemented on a Windows NT/95 platform.

Acknowledgments

Foremost, I would like to thank my advisor, Professor Tore Risch for giving me a chance to work on such an exciting project. His expertise and exuberant enthusiasm were of great help in shaping the achievements of this work. I am also grateful to the other present and past members of the EDSLAB research group at Linköping University for valuable advice and discussions. Special thanks go to Timour Katchaounov who implemented and evaluated (and also named) the decomposition tree distribution. It was not as easy a task as we expected it to be. Gundars Kulpus implemented the type importation and the proxy types hierarchy definition. I also thank Jörn Gebhardt for the careful proof-reading of the query decomposition description.

I thank my closest family for the generous support and gentle care during my twenty-three years of education. This thesis concludes not only the biggest project of my life, but probably the biggest project of my mother's life too. My father has always been there to put me back on track when I strayed. My grandfather Jonče Josifovski inspired me to pursue a carrier in science by telling me those fantastic stories as a young boy. I am also grateful to the Erma family for accepting me as one of their own, especially to my Milli who endured so much stress in the course of this work.

This work was funded by ECSEL, the Excellence in Computer Science and Engineering in Linköping Program.

To my grandfather Ivan Pendarovski - Vančo,
for his love and care

Contents

1 Introduction **1**

2 Data Integration by Multidatabase Systems **5**
2.1 Enabling technologies 5
2.1.1 Database systems 6
2.1.2 Networking technologies 9
2.1.3 The object-oriented paradigm 10
2.2 A taxonomy of the data integration research 12
2.2.1 Global schema systems 14
2.2.2 Federated architecture 15
2.2.3 Multidatabase languages 16
2.3 Autonomy of the data sources 16
2.4 Data and schema heterogeneity 17
2.5 Query processing and data integration 20

3 An Overview of the AMOS*II* System **23**
3.1 Data model 24
3.2 Query language 26
3.3 Query processing in AMOS*II* 28

4 Data Integration by Derived Types **35**
4.1 Object-oriented view system design 36
4.1.1 Derived types 36
4.1.2 Generation of OIDs for the DT instances 38
4.1.3 Derived types and inheritance 40
4.1.4 Derived subtyping language constructs 41
4.2 Querying derived types 43
4.2.1 Overview of the derived types implementation 44

4.2.2 Proxy types and objects 45
4.2.3 DT extent function and template 48
4.2.4 Generation of OIDs for DT instances 55
4.2.5 Processing of queries using locally stored functions . . 56
4.2.6 The Transformation algorithm 59
4.3 Database updates and coercing 61

5 Integration of Overlapping Data 63
5.1 Integration union types 65
5.2 Modeling and querying the integration union types 68
5.2.1 Late binding over derived types 70
5.2.2 Normalization of queries over the integration union types 73
5.2.3 Managing OIDs for the IUTs 75
5.3 Performance measurements 77

6 Query Decomposition and Execution 87
6.1 Query decomposition 88
6.1.1 Data source types and functions with multiple implementations 90
6.1.2 The predicate grouping phase 93
6.1.3 MIF predicates execution site assignment 97
6.1.4 Cost-based scheduling 105
6.1.5 Decomposition tree distribution 112
6.2 Object algebra generation and run-time support 117
6.2.1 Object algebra generation 117
6.2.2 Inter AMOS*II* communication and the SAE operator . 119

7 A Survey of Related Approaches 131
7.1 Multidatabase systems 132
7.1.1 Disco 132
7.1.2 Garlic 134
7.1.3 Pegasus 136
7.1.4 TSIMMIS 140
7.1.5 Multibase 141
7.1.6 Data Joiner 142
7.1.7 MIND 143
7.1.8 IRO-DB 146
7.1.9 DIOM 149

7.1.10 UNISQL 152
7.1.11 Remote-Exchange 152
7.1.12 Myriad 153
7.2 Object-oriented views 155
7.2.1 Multiview 155
7.2.2 O2 Views 157

8 Summary and Conclusions **159**

A Abbreviations **163**

References **165**

List of Figures

2.1 An MDBMS reference architecture 13

3.1 Interconnected AMOS*II* servers 24
3.2 Query processing in AMOS*II* 28
3.3 Two algebraic representations of the example query 32

4.1 Integration by derived types (subtyping) 37
4.2 Integration by integration union types (supertyping) 40
4.3 Placing the proxy types in the type hierarchy 46

5.1 An Object-Oriented View for the Computer Science Department Example 66
5.2 IUT implementation by ATs 69
5.3 Query: select salary(e) from csd_emp e; 79
5.4 Query: select salary(e) from csd_emp e where ssn(e) = 1000; . 82
5.5 Selecting salary for the CSD employees with and without range selection (salary(e) > 2000) 83
5.6 a) Queries with locally materialized functions over IUTs. b) Queries calling several derived functions over IUTs. 84
5.7 Comparison of execution times over a 10Mb network with an ISDN network. 85

6.1 Query Decomposition Phases in AMOS*II* 89
6.2 Data source capabilities hierarchy 92
6.3 MIF predicate site assignment heuristics 99
6.4 Query graph grouping sequence for the example query 101
6.5 Case 5 example and the possible outcomes 102
6.6 A query processing cycle described by a DcT node 107
6.7 Two decomposition trees for the example query 109

6.8 Two tree generation rules: a) adding a local SF to a partial tree, b) adding a remote SF to a partial tree 111
6.9 Node merge: a) the original tree b) the result of the merger operation 114
6.10 Execution diagrams of the decomposition tree of the example query before node merge and after 115
6.11 Project-concat SAE implementation 123
6.12 SAE by semi-join 125
6.13 SAE by semi-join and a temporary index 128

Chapter 1

Introduction

An important factor of the strength of a modern enterprise is its capability to effectively store and process information. As a legacy of the mainframe computing trend in the previous decades, large enterprises often have many isolated data repositories used only within portions of the organization. The methodology used in the development of such systems, also known as *legacy systems*, is tailored according to the application, without concern for the rest of the organization.

While these systems contributed to faster development of the companies in the past, their inability to interoperate and provide the user with a unified information picture of the whole enterprise is a "speed bump" in the process of taking the corporate structures to the next level of efficiency. This phenomenon is exemplified in the new international corporations build by global mergers. The informational assets of such companies are both geographically and structurally far apart.

The recent development of the network technology provided the corner stone for the integration of legacy systems. Faster network technologies bridged the physical gap between these systems. Nevertheless, this did not eliminate the burden of accessing the legacy systems in their diverse native formats. A study of the data processing patterns of the Fortune 500 companies conducted at the beginning of the 90s [40] has shown that over 80% of the surveyed companies accessed data in multiple systems.

Another recent trend is that dumb terminals as access points are replaced

by more powerful workstations having substantial processing capabilities, but which are nevertheless too small to hold all the data that a user might need. The power and the network connection capability of these workstations can vary considerably in a large enterprise: from a stationary workstation with a network connection of few million bits per second, to hand held devices with network links of only few thousands bits. It would be a great expense for a company to adjust the corporate software for these different circumstances. A solution is to have an adaptable system that can take advantage of the different configurations without changing the software implementation.

Users, on the other hand, require simple and fast access to the information. "Simple" usually means that the picture of the data in the enterprise corresponds to the user's view of the enterprise and its position in it. Mainly, this translates into three technical requirements:

1. *Location transparency*: the user is not aware of the physical location of the data. The data access is uniform regardless of whether the information is stored locally in the user's workstation, or in a systems half way around the globe.

2. *Heterogeneity transparency*: some of the legacy systems might provide equivalent, complementary or conflicting information. In such cases, the data must be reorganized, so that the user gets a picture of the data where the redundancies are eliminated and conflicts are resolved.

3. *Autonomy*: The existing systems and applications should function as before, without any modification and dependencies to the added integration framework.

In the last two of decades, research in the area of *data integration* has contributed several classifications of the challenges in this area, and proposed a number of solutions. The *wrapper-mediator* approach, described in [85], divides a data integration system into two functional units. The wrapper provides access to the data in the *data sources* using a *common data model* (CDM), and a *common query language* (CQL). The mediator provides a semantically coherent CDM representation of the combined data from the wrapped data sources.

This thesis presents a design, an implementation, and an evaluation of a *mediator database system* named AMOS*II*. In this system, the problem of data integration is tackled from a *database* perspective. The database field is one of the better established fields within computer engineering, with a

well-developed theoretical background and a large number of commercial products that have benefited from it. The main objective of database research is to explore how to store and query large amounts of data. The AMOS*II* project adapts and combines some results of the database research with an array of novel ideas in order to tackle the data integration problem. The data integration functionality of AMOS*II* is provided by two conceptual units:

- An *OO database view system* provides a coherent and unified view to the data integrated by the mediator.
- A *multidatabase query processing engine* processes the queries over data in multiple repositories.

The rest of this thesis is organized as follows. Chapter 2 introduces the field of data integration and places the presented work in the context of related research. It also introduces the basic terminology and gives a more precise definition of the data integration problem. The basic features of the AMOS*II* system that are used as a basis for the work in this thesis are presented in chapter 3. The OO view system is described in chapters 4 and 5. Chapter 4 introduces the basics of the OO view system for data integration in AMOS*II*. Chapter 5 extends the concepts in chapter 4 to integration of sources with overlapping data. The multidatabase query processing facilities are presented in chapter 6. A detailed comparison of AMOS*II* with some other related research prototypes and commercial products in the area is presented in chapter 7. A summary is given in chapter 8.

Chapter 2

Data Integration by Multidatabase Systems

The term *multidatabase management system* (MDBMS) implies a system consisting of several databases. In the literature, this term has been mainly used to describe systems providing various degrees of integration and interoperability among a number of databases or other types of data sources. We note here that, in spite the implications of the name, one of the most important goals of MDBMS research is to encompass data sources that are not databases. Nevertheless, the databases remain one of the most important type of data sources. This chapter introduces the research in MDBMSs by first giving an overview of the technologies that contributed to its proliferation. Next, a short taxonomy of the data integration research encompassing the field of MDBMSs is presented. Finally, some issues characteristic for MDBMSs are discussed.

From the various overviews of the multidatabase research field, for an interested reader we single out [69], [5] and [7], also used in the preparation of this chapter.

2.1 Enabling technologies

MDBMS research emerged as a result of the advances in several related disciplines and by increasing sophistication of the users' demands. This section provides a short overview of the most influential areas from the aspect of the work presented in this thesis.

2.1.1 Database systems

A database represents a collection of information managed by a database management system (DBMS). The DBMS allows the user to [83]: create new databases and their logical descriptions named *schemas*; store securely large amounts of data; query and modify the database using a *query language*; and control the simultaneous access to the data by a multitude of users.

Databases and DBMSs play a major role in almost all areas where computers are used. The first commercial DBMSs, developed in the 1960s, came into existence when the complexity of the applications dealing with large amounts of data could not be efficiently satisfied by the file system services. Since then, typically the development of DBMS technology has been classified by the methodology for describing the schema and the data, named *data model*. Different models have emerged in the last four decades. Examples of the earlier models are the hierarchical and the network data models in which the data in the database is represented in the form of a graph. The query languages for querying this data allow the user to navigate through the data graph. Writing such navigational programs is hard. Also, the graph representation used in the queries closely followed the physical layout of the data on the storage device. Any change in the storage patterns require changing the applications.

In the early 1970s the *relational data model* was proposed in [10]. This model has been the single most influential idea in the development of the DBMSs to date. According to this model, the user views the data in the database in the form of simple *tables*, consisting of one or more labeled *columns*. The table entries are named *rows* or *tuples*. Each column of each row contains a *value* that can contain a number, string of characters, or other simple concept from the real world. A special $NULL$ value specifies the absence of a user-supplied value.

This conceptual view of the data in the relational model is close to many of the traditional, non-electronic data representations. The DBMS, however, can internally store the data in more complicated structures that allow faster access and manipulation of the data. A change of the storage structures does not change the queries, specified in a formalism named *relational calculus*. There are a few languages that provide a "friendly" syntax for the relational calculus, the most widely used of which is IBM's Structured Query Language (SQL). The relational calculus and the SQL languages are *declarative*, i.e. the user only specifies *what* is to be retrieved from the database and not *how*

is it retrieved. Therefore, the relational calculus is used to state non-ordered descriptions of the user's queries. In order for the DBMS to execute the query, it must translate this to a program that precisely describes how the data is retrieved from storage. These programs in the relational DBMSs are usually described by a formalism named *relational algebra*. Some of the more typical relational calculus and algebra operators are:

- **selection (σ)**: selects a subset of the input table based on a condition (e.g. all employees that have salary larger than a certain amount)
- **projection (π)**: selects a subset of the columns of the input table (e.g. selects the salaries from an employee table containing the employee names and salaries)
- **join ($\bowtie$)**: produces a new table by matching the tuples of two input tables by given conditions (e.g. returns a table containing the names, salaries and social security numbers of employees, by matching the rows of two tables one containing the names and the social security numbers, and another containing the salaries and the social security numbers)

These three operators are considered the basis of any relational DBMS *query processor*. Queries composed of these three operators are named *select-join-project* queries.

The process of translation of the calculus queries into relational algebra programs is called *query processing*. Typically, a calculus expression translates into many equivalent algebra expressions, also named *query execution plans* (QEPs), that all produce the same result, but use different orderings of the operators and algorithms for their evaluation. Different plans often require different execution times that may vary by several orders of magnitude. Consequently, it is of great importance that the DBMS chooses a plan with a low execution time. The process of selecting a plan with as low as possible an execution time is named *query optimization*. The query optimization is one of the most critical and complex phases of query processing.

Although the relational data model, calculus and algebra solved many of the problems present in the previous approaches, almost three decades of usage has exposed a number of limitations. The research community reacted to this by developing a row of post-relational models and approaches. One of the more successful has been the *Object-Oriented* (OO) approach. Systems

that provide an OO data model and a declarative query language have recently been named *Object-Relational* (OR). AMOS*II* is one such system. A brief overview of the OO data model is presented later in this section.

The users of a database can each require different data representations and queries. The design of the database schema should aim to satisfy the representational needs of the majority of the users. Nevertheless, this is not always possible and a database schema might be suitable for some of the users, but compel others to write long and tedious queries in order to obtain the result in the required format. The *database view* mechanism alleviates this problem by allowing definition of virtual schemas for different users. These are defined using stored query specifications that transform the data from the stored format to the format required by a user. To the user, the view is transparent and has the appearance of an ordinary schema. Queries over the views are translated by the DBMS into queries over the database schema. Views are a well established technology, present in almost every commercial relational DBMS. Views in OO and OR systems have been a subject of intense research in the last decade and the first commercial products are starting to emerge.

Another popular dimension of classification of the DBMSs is their architecture. Here, one of the classifications is into *centralized* and *distributed* systems. The former type represents systems where all the data is stored in a single repository and all the accesses are processed by a single DBMS. The technically more advanced distributed DBMSs store the data in multiple repositories and access it by a cooperating set of DBMSs. A distributed architecture provides improved performance, reliability and availability, but has an increased complexity compared to the centralized one.

This thesis presents a *distributed multidatabase architecture* for data integration, that has it origins in the distributed DBMS approach, but differs from it by not assuming homogenous, cooperating systems providing a uniform interface. Furthermore, the data in a distributed database is distributed, stored and updated under the strict control of the DBMS. This thesis explores a system that has no control over these issues.

There is an extensive literature on the subject of databases and DBMSs, to name just a few of the more popular text books: [83, 18, 72]. A classical textbook on distributed databases is [61].

2.1.2 Networking technologies

The technological limitations of the size and the complexity of a single system led to the development of the computer networks. A computer network is defined in [61] as: *an interconnected collection of autonomous computers capable of exchanging information.* This definition states two main requirements: that the systems are *interconnected*, i.e. that they can exchange information, and that the systems run their own programs in an *autonomous* manner. Besides the computers, often referred to as *nodes* or *sites*, the network also contains communication links and specialized network traffic management equipment to increase efficiency and manageability.

The interaction of the network sites can be modeled by different paradigms. Two of the most popular are *peer-to-peer*, where nodes treat each other as equals, and *client-server* where the clients send requests for processing to the servers that return the replies to the calling clients. In multitasking systems, a site in a network is not necessarily a physical computer, but it can rather be represented by a single *process* running on a computer. Therefore, more than one logical site can reside on a single physical site.

The development of networking technology has been one of the most influential factors in the rapid growth of the computing industry in the last two decades. This development has shifted the accent from development of isolated centralized systems to connected and distributed decentralized systems. The impact of this shift can be seen in an array of new computer network-based products that have changed the world, such as the Internet and digital mobile telephony, that are two of the most dynamically developing fields in the area.

Although there are many parameters that illustrate the advances of computer network technology, the two most commonly used are the increase in the availability of network connections and their capacity [39]. While only a decade ago the most common network connection to an end user was 9600 bits/second (baud), today's local area networks (LANs) easily reach the 10^9 baud mark. 128 Kbaud Integrated Services Digital Network (ISDN) connections are readily available in almost all households and offices in the developed countries. In the coming years, technologies such as the broadband network standard named Asynchronous Transfer Mode (ATM) will increase these limits by some orders of magnitude. This technology is available for both local and wide area networks and will provide services with a bandwidth of up to 150 Mbaud. In digital mobile telephony the bandwidth is

still largely limited (e.g. 9600 baud for the GSM standard), but the newly announced standards such as the Wireless Collision Detection Media Access (WCDMA) developed by a few European vendors will lift this limitation to around 2 Mbaud.

A classical textbook on computer networks is [79].

2.1.3 The object-oriented paradigm

While the relational model proved successful in business applications, it proved to be inefficient in the support of applications as CAD/CAM systems, office automation and scientific databases. These applications require a more complex structure of the data, longer-transaction duration, new data types (e.g. multimedia, matrices, documents), and non-standard application-specific operations [18].

The Object-Oriented (OO) model was developed to cope with these requirements. The origins of this model are in the OO programming languages that started with the language SIMULA in the late 1960s. Since then, a variety of research prototypes and, to some extent, commercial database product have adopted this model. As opposed to the relational model, there is still no widely accepted standard for an OO data model and query language. Two most notable efforts are the Object Database Management Group (ODMG) standard [9], and the OO version of the SQL language standard SQL3 [74].

In the OO model, the real world entities are modeled as *objects* classified into *classes*. The objects belonging to a class are called *class instances*. The set of all class instances make the class *extent*. The designer has the capability to model both the structure of the objects as a set of *attributes* (roughly corresponding to the table columns in the relational model), as well as operations (or *methods*) that can be performed over the objects of a particular class. The methods are specified in a procedural or declarative language. The set of all attributes and methods applicable to the objects of a certain class is the *interface*, or *behavior* of that class. The designer can also specify that some of the attributes/methods can be used only internally within other methods, exposing to the user of the objects only a part of the interface. This technique is called *encapsulation* and provides for increased maintainability of the code.

As opposed to the relational model where the tuples of interest are identified by unique combinations of their column values (*keys*), in the OO model each object is assigned by the system an immutable, unique *object identifier*

(OID). The OIDs can be used by the user to directly access the object instance. They can also be stored as attribute values of other objects. While in the relational database, all the data accesses are performed by relational calculus queries, in an OO database the objects can be accessed by *navigating* through the graph of objects connected by edges of OID as attribute values. We note here that as opposed to the network data models where the links are physical pointers, the OIDs are logical pointers independent of the physical storage implementation.

Another important feature of the OO model is *class inheritance*. This mechanism allows a class to be defined as a *subclass* of another class, named *superclass*. The subclass inherits all the attributes and methods of the superclass, and can also define its owns. The directed graph of the classes and the inheritance dependencies is usually called *class hierarchy*. Some of the systems support *multiple inheritance* where a class can inherit from more than one superclass. The usual semantics is that an instance belonging to a class, also belongs to all of it superclasses (extent-subset semantics).

The class hierarchy and the extent-subset semantics set a stage for *polymorphic behavior* of the class instances. This means that a method can have different implementations for different instances of a class, depending on which of its subclasses the instance belongs to. Inversely, multiple implementation definitions are allowed for a single method in different classes. When such a method is invoked over a set of instances of that class, the system invokes the most specific implementation. For example, let's assume that a class `shape` is defined with two subclasses `circle` and `square` that define a method *area()* calculating the area of a particular shape. When the method *area()* is invoked over a set of shapes, the circles should be processed by the implementation defined for circles, while the squares with the implementation defined for squares. The instances of the type `shape` exhibit in this case non-uniform (polymorphic) behavior. Polymorphism requires that method implementations are chosen during run-time, when the query or the program is executed. The mechanism that allows this is hence called *late binding*, as opposed to *early binding* where the method implementation is chosen during compile time.

2.2 A taxonomy of the data integration research

Research in the field of data integration systems has identified two basic approaches. One uses *eager materialization* of local copies of the queried data from the data sources, trying to reduce the response time by performing most of the costly operations before the query is issued [34]. The other approach, which we name *passive*, fetches the required data when it is requested. Which of the two approaches yields better results depends on factors such as available resources, the size of the data, and the query and update frequencies.

Eager materialization has an advantage when, for example, the update frequency of the data is low, the sources support active mechanisms for propagating the changes to the data integration system, and the data integration system has available resources to maintain the materialized data. A variant of the eager approach, *data warehousing*, performs a materialization of all the data from the sources that might be needed in the user queries in advance, before the queries are executed. In this way, queries that do not have strict currency requirements, in the presence of adequate resources, can be executed over local copies of the data. Another variant of the eager materialization uses *active maintenance* of the local copies by incrementally applying the changes of the original data to the copies. This variant is adequate when the change rate is low, and the data sources can provide means for actively propagating the changes.

By contrast, the passive approach has advantages when the user's system is too small to host the materialized data that he queries, or when the maintenance of the materialized copy is too costly to perform (e.g. because of large volume of updates). Also, the passive approach is less intrusive towards the autonomy of the data sources. It has been identified that both approaches are important and complementary to each other [87] [34].

The work presented in this thesis is based on the passive approach. Accordingly, the rest of this section discusses data integration architectures based on the passive approach, that are also the data integration systems most often associated with the term "multidatabase management systems" as defined above. In [69], a reference MDBMS architecture is presented (Figure 2.1). This architecture is based on mappings between schemas on 5 levels:

- **Local schema** A local schema represents the data in a data source. There is one local schema for each data source. The local schemas are expressed using a local data definition language and a local data model, if such exist. Non-database data sources might describe the local data

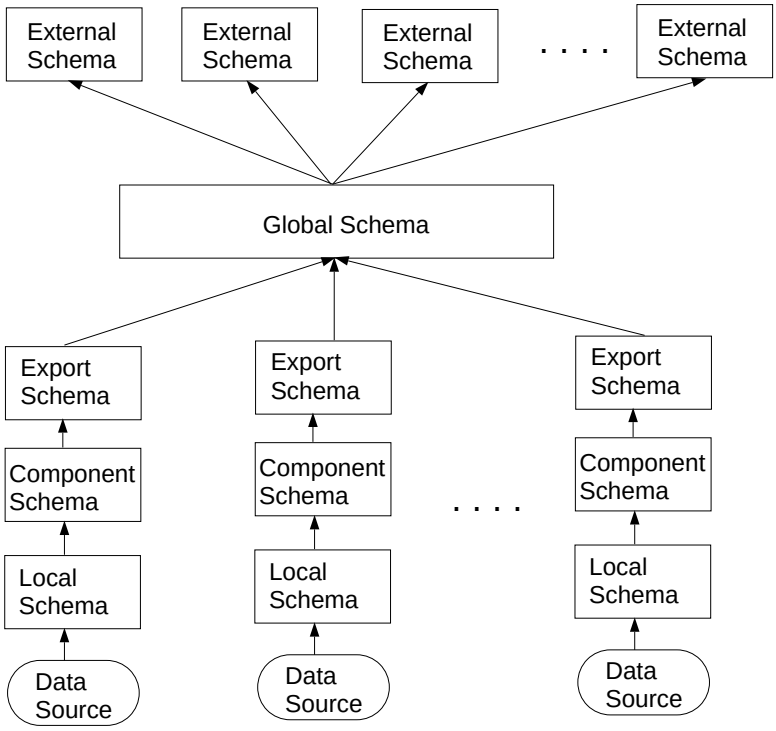


Figure 2.1: An MDBMS reference architecture

and its organization using other formalisms.

- **Component schema** A component schema is a CDM representation of a local schema. The local schema is translated into a CDM representation if the CDM is different than the local data model, otherwise the local and the component schemas are the same.

- **Export schema** In some architectures, each data source decides the portion of the data that is going to be available for non-local access. The export schema models the portion of the component schema visible non-locally. It is also expressed in the CDM.

- **Federated or Global schema** A federated (global) schema is an integration of all the export schemas. Depending on the particular framework applied, this schema can be called either global or federated.

The term global schema is used when there is only one such schema. There can be more than one federated schema.

- **External schema** An external schema represents a subset of the global schemas tailored for a particular user or group of users.

Depending on the level of integration, MDBMSs can be classified into 3 categories [5]: *global schema systems*, *federated databases* and *multidatabase language systems*. These categories reflect design efforts to accommodate the conflicting requirements of achieving an efficient and usable system by larger level of sharing on one side, and preserving the autonomy of the data sources, on the other. On the one extreme of this spectrum are systems that are close to the distributed databases in building a global integrated schema of all the data in the sources. The opposite side represents systems that provide just basic interoperation capabilities and leave most of the integration problem to the user. The rest of this section overviews the features of each of these categories.

2.2.1 Global schema systems

Historically the first approach to building an MDBMS is the approach where the export schemas of multiple databases are integrated into a single global view (schema). According to the reference architecture, the export and component schemas are equal and there is a single global schema. The user is not aware of the distribution and the heterogeneity of the integrated data sources. Furthermore, if the schema does not change frequently, it can be stored locally, at the client, for faster access. Nevertheless, this approach has been shown to exhibit the following problems [5]:

- Since the general problem of integrating even only two schemas is undecideable, the process of integration of multiple schemas is very hard to automate. Global schema integrators must be familiar with all the naming and structure conventions of all the data sources and integrate them into a cohesive single schema without changing the local schemas.

- There are two basic approaches to integrating the component schemas into a global schema. In the first, the component schemas are integrated pair-wise. A hierarchical application of the integration leads to a schema integrating all the component schemas. The other approach is to integrate all the component schemas at once. Both approaches

have problems. The first one could produce different results when different integration orders are used, while the other one is usually too difficult.

- It is necessary for the component databases to reveal some information about the semantics of their data for this type of schema integration to be possible. This violates the autonomy of the data sources.

2.2.2 Federated architecture

In the federated MDBMS (FMS) the export schemas are only a subset of the component schemas. Each data source is given control over the portion of the data that will be exported. The federated schema does not need to be an integration of all the export schemas. It can integrate only portions of the export schemas of interest to the users using the federated schema. More than one federated schema can be defined according to the users' requirements. Each user can then further refine its export schema to fit his own requirements. The wrapper-mediator approach used in this work is a variant of the federated architecture approach.

In a *tightly coupled FMS*, the mappings between the different schemas is kept in a federation directory, accessed during the processing of the queries over the federated schemas. Maintaining the directory creates an overhead in this type of system. The size of the directory can grow dramatically as the number of data sources and users increase. It can also become a performance bottleneck when accessed by a large number of users. These problems are reminiscent of the problems of maintaining a global schema described above.

Loosely coupled systems do not have a centralized directory. The user creates and maintains his own integrated schema in the form of a local view. The maintenance problems noted above disappears. A possible drawback of this approach is that more than one user might need to perform the same view modeling, without the possibility of reusing the definitions. Furthermore, a change in an export schema affects all the users who have a view dependent on it.

A solution to the problems noted above is to allow a gradual transition from the federated into export schemas by a hierarchy of small intermediate schemas. This approach breaks the repository into smaller and more maintainable units, while allowing reuse of the view specification and modularity in the view definition and change. Because of these advantages, this is the approach of choice for the design of the system presented in this thesis.

2.2.3 Multidatabase languages

This approach does not provide any type of global schema. The only means of accessing the data in the data sources is by language primitives for specification of queries over data stored in multiple sources. In the reference architecture, this means that the user explicitly sees all the component schemas of the integrated sources. The multidatabase language approach is usually used for integration of data sources that are databases.

An important feature of the multidatabase languages are constructs that allow for iteration over the meta-data of the local and remote databases. Queries can be defined that iterate over all known databases, or a set of tables in the databases based on some regular expression [51]. These are translated into multiple queries that are executed by the relevant systems. Because, the operations performed over the local tables need not be the same in all databases, the translation process is capable of generating queries that use different operators in different systems in order to construct the required result (e.g. some database might contain the requested result in a single table, while others in a set of tables that need to be joined first). There are limited constructs for resolution of naming, scaling and unit discrepancies of the data in the data sources by user defined expressions.

The main criticism of the multidatabase language approach is the low level of transparency provided to the user. The user is responsible for finding the relevant information, understanding each database schema, detecting and resolving the semantic conflicts, and finally, building the required view of the data in the sources [5]. The advantages of the approach are that it is not intrusive against the autonomy of the data sources and there is no global/federated schema maintenance and access overhead.

2.3 Autonomy of the data sources

As opposed to the distributed DBMSs where the nodes are under the control of a single authority, the autonomous data sources treat the MDBMS only as another client. The requests to the data sources are performed using the interfaces available for the clients. More specifically, the autonomy of the data sources can be classified in several different categories [69, 86]:

- **Design Autonomy**: The data source manager decides what data is stored in the database and how is it stored and interpreted. This in-

cludes the choices of data model, query language, database constraints, etc.

- **Communication Autonomy**: The data source decides which requests it will answer and when it will answer them. In other words, the services provided to the MDBMS are decided by a local database manager.

- **Execution Autonomy**: The MDBMS cannot make any assumptions about the algorithms and methods used in the data sources to process the requests. The execution strategy is decided locally in the data sources. Also, no assumptions can be made about the relative order of execution of concurrent requests.

- **Association Autonomy** The data sources decide how much of its data and processing capabilities it will share with the MDBMS. It can limit the access to only a portion of all the available data. The query requests can be limited to certain types of operations (e.g. projection, selection and join) or certain functions (e.g. matrix addition and multiplication). Furthermore, the sources are not obliged to expose internal data as, for example, statistical data or execution time estimates.

2.4 Data and schema heterogeneity

One of the major challenges in integrating multiple heterogeneous data sources is in *understanding* and *translating* the data from all the data sources into a common context [5] . The main difficulty in this process is the presence of *semantic heterogeneity* among the data and meta-data (schema) in the different data sources. A data item in one data source can correspond, complement or conflict with data in the other data sources. In order to present the user with coherent view of the data in the sources, the system needs to provide some means of *reconciliation* of the semantic heterogeneity.

The most cited cause for semantic heterogeneity is the design autonomy of the data sources. To illustrate such a case, we consider an example of two databases storing the salaries of the employees of a company formed by a merger of one Swedish and one US company. The Swedish company database stores the salary amounts in crowns, while the database in the US stores the amounts in dollars. A user presented with, for example, two

salaries for the same person working in both countries cannot easily perceive the exact amount of the person's salary in the local currency.

In order to design a system with reconciliation facilities, first a classification of the possible semantic heterogeneity is needed. The literature provides several such classifications, three more extensive of which that take into account an OO data model are [44], [70] and [31]. In the rest of this section, a short summary of the classification in [31] is presented. In this classification, the semantic heterogeneities are first divided into three groups:

- Heterogeneities between **Object classes**
- Heterogeneities between **Class structures**
- Heterogeneities between **Object instances**

The heterogeneities between object classes are further classified into differences in:

- **Extents**: (i) the extents can represent different parts (entity sets) of the real world (e.g. two classes representing colors can have different numbers of colors in them); (ii) the intersection of the extent can be anything from equal to both the extents, to an empty set; (iii) an extent of a class in one source might correspond to the extents of several classes in the other sources, etc.
- **Names:** (i) same name can be used for different concept (homonyms); (ii) the same concept can be named differently in different data sources (synonyms).
- **Attributes and methods:** (i) the absence of a method or an attribute; (ii) arity differences; (iii) attribute constraints differences (e.g. minimum/maximum value, NULL value, minimum/maximum arity, uniqueness, etc.)
- **Semantics and syntax of domains:** (i) semantic domain differences include differences in the internal OID formats of the data sources, differences in the key values of corresponding class entities due to different coding of the keys, differences in dimensions, units and scale, etc. (ii) the syntactic domain differences are in the coding ranges, the length of the literal types, character/numerical differences, coding of dates, etc.

- **Constraints** besides the already mentioned simple constraints, each of the data sources can enforce different complex constraints based on more than one class in the schema.

The class structure heterogeneities are divided into:

- **Generalization/Specialization inconsistencies:** two corresponding classes might have a different number of super-/sub- classes, or the subclass membership can be over different criteria (e.g. a class *truck* subclass of class *vehicle* can be populated using different criteria in different data sources).

- **Aggregation/Decomposition inconsistencies:** based on the properties of the object graph represented by the objects, the navigational links, their arity, and interactions. Three types of aggregations (i.e. interactions among objects and their attributes) are defined: a simple aggregation, where the object does not depend on its attributes; composition aggregation, where the attribute must have a value for the object to exist (e.g. keys); and collection aggregation where the attribute can be multivalued. This class of inconsistencies deals with cases where the corresponding attributes in different data sources are of different aggregation types, or have different constraints on their value sets in the case of a collection aggregation.

- **Schematic Discrepancies:** some concepts represented as data in one of the data sources are represented as meta-data in another. For example, one relational source might contain tables *cars* and *trucks*, while another models the same concepts using a single table named *vehicle*, and an attribute in this table to distinguish between the types of vehicles.

Finally, the object instances heterogeneities are classified into:

- **Presence/Absence:** an object of a class in one of the sources has no corresponding object in the corresponding class in the other source.

- **Discrepancies in attribute arity:** the corresponding multivalued attributes of two corresponding objects from different data sources have different arities.

- **Value Discrepancy:** the corresponding attributes of two corresponding objects from different data sources have different values.

The integration framework presented in this thesis will mostly concentrate on resolving the semantic heterogeneity of object classes and object instances among the data sources. Nevertheless, its flexible structure allows for extensions that would cover most of the other heterogeneities.

2.5 Query processing and data integration

One of the reasons for the success of the database technology is the capability of the DBMSs to accept *declarative* query requests from the user. As noted earlier, the user only needs to specify *what* is to be retrieved, rather than *how* it is retrieved. In other words, queries are not programs stating precisely how the data is retrieved. The burden of making a query execution plan from a query is taken by the DBMS. In an multidatabase environment consisting of heterogeneous and autonomous data sources, this task becomes even more demanding.

Resolving heterogeneity usually requires advanced queries containing operators that are more complex than in the traditional select-project-join queries. An example of such an operator, used in this work to integrate overlapping data from different sources, is the *outer-join* operator. This operator returns not only the matching tuples of the operands, but also the non-matching tuples, padded by NULL values. This operator does not have the associativity and commutativity properties used heavily in optimization of regular join-based queries.

Another issue is the difference in the capabilities of the participating data sources. While in the distributed database framework all nodes have the same functionality, here some nodes might not even be databases (e.g. an e-mail system). This makes the query compilation and the division of the tasks among the nodes harder than in distributed databases.

The autonomy of the data sources also greatly influences the query processing in an MDBMS. As the MDBMS interacts with the data sources only via an external interface, the internal statistical information needed for the query optimization is not available. Obtaining this type of information is typically very hard in an MDBMS operating over autonomous sources. In this thesis we do not elaborate on this problem. A few solutions to the problem have been proposed in the literature: *query sampling* in [88], *query probing and piggyback* in the same reference, and *calibration and regression* in [31]. A survey of these techniques is presented in [5].

The MDBMS environment is also much more dynamic in comparison with the classical distributed database environment. Here, the participating data sources are free to withdraw from the system or refuse certain requests.

Chapter 3

An Overview of the AMOS*II* System

The AMOS*II* system was developed from the AMOS system which has its roots in the workstation version of the Iris system, WS-Iris [52]. The core of AMOS*II* is an open, lightweight, and extensible database management system (DBMS). The aim of the AMOS*II* architecture is to provide for efficient integration of data stored in different repositories by both active and passive techniques. To achieve better performance, and because most of the data resides in the data repositories, AMOS*II* is designed as a main-memory DBMS. Nevertheless, it contains all the traditional database facilities, such as a recovery manager, a transaction manager, active rules, and an OO query language. A running instance of AMOS*II*, named an AMOS*II* server (or simply server), provides services to applications, as well as to other AMOS*II* servers.

Figure 3.1 illustrates the different roles that an AMOS*II* server can assume. In this example, several applications access data stored in several data sources through a collection of interconnected AMOS*II* servers. AMOS*II* servers can run on separate workstations and provide different types of data integration services. One server is designated to be a *name server* and provide information about the locations of the servers on the net. Different interconnecting topologies can be used to connect the servers depending on the integration requirements of the environment. Also, a single AMOS*II* server can perform more than one task described in the figure and serve more than one application simultaneously. Each AMOS*II* is a fully fledged DBMS and

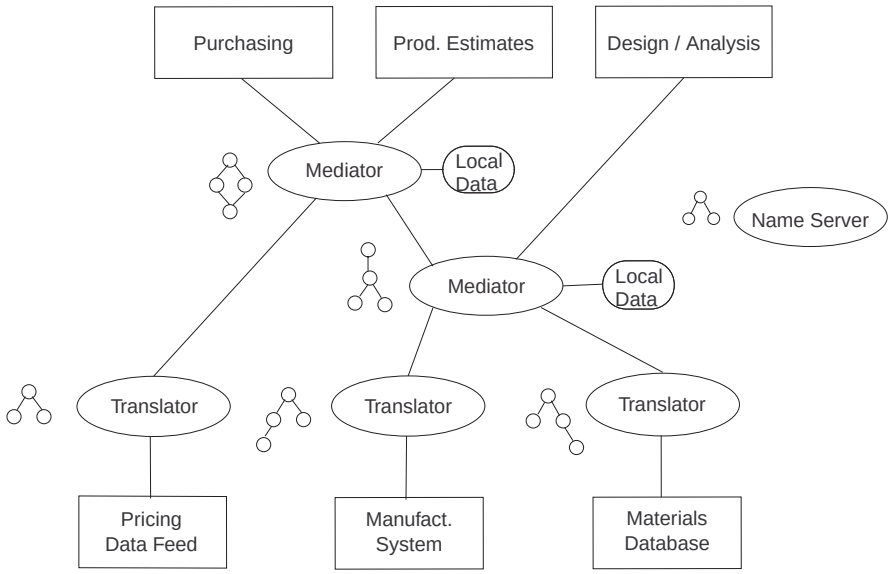


Figure 3.1: Interconnected AMOS*II* servers

can store data locally. Imported and local data is described in each AMOS*II* by an OO type hierarchy.

In [23], an approach to wrapping relational data sources with AMOS*II* is described. Here, the sources are not only wrapped, but also some query optimization techniques are used to simplify the queries on both local and relational data. Therefore, to distinguish between the wrapper subsytem in AMOS*II*, and an AMOS*II* server having the role of wrapping a data source with this extended functionality, the second is named *translator*. The term wrapper will be used to represent the wrapper subsystem.

This thesis describes the design and implementation of the mediation services in AMOS*II*.

3.1 Data model

The data model in AMOS*II* is an OO extension of the DAPLEX [71] functional data model. It has three basic constructs: *objects*, *types* (i.e. classes),

and *functions*. Objects model entities in the domain of interest. An object can be classified into one or more types which make the object *instances* of those types. The set of all instances of a type is called the *extent* of the type. Object properties and their relationships are modeled by functions.

The types in AMOS*II* are divided into *literal* and *surrogate* types. The literal types, e.g. *int*, *real* and *string*, have a fixed (possibly infinite) extent and self-identifying instances. Each instance of a surrogate type is identified by a unique, system-generated object identifier (OID). The types are organized in a multiple inheritance, supertype/subtype hierarchy that sets constraints on the classification of the objects. One example of such a constraint is: If an object is an instance of a type, then it is also an instance of all the supertypes of that type; conversely, the extent of a type is a subset of the extents of its supertypes (extent-subset semantics). The AMOS*II* data model supports multiple inheritance, but requires an object to have a single most specific type.

The surrogate types are divided into *stored*, *derived*, *proxy*, and *integration union* types:

- The instances of *stored* types are explicitly stored locally in AMOS*II* and created by the user.
- The extent of a *derived* type (DT) is a subset of an intersection of the extents of the *constituent* supertypes. The instances of the supertypes are selected and matched using a declarative query. DTs are described in chapter 4.
- The *proxy* types represent objects stored in other AMOS*II* servers or in some of the supported types of data sources. The proxies are also described in chapter 4.
- The *integration union types* (IUTs) are defined as supertypes of other types. An IUT extent contains one instance for each real-world entity represented by the (possibly overlapping) extents of the subtypes. The integration union types are the subject of chapter 5.

The functions are divided by their implementations into three groups. The extent of a *stored* function is physically stored in the database. *Derived* functions are implemented in a declarative OO query language AMOSQL. *Foreign* functions are implemented in some other programming language, e.g. Lisp, Java or C++. Each foreign function can have several associated access

paths having different implementations and, to help the query processor, each access path has an associated cost and selectivity [1] function [52]. This mechanism is called a *multi-directional foreign function*.

3.2 Query language

The AMOSQL query language is based on the OSQL [53] language with extensions of mediation primitives, multi-directional foreign functions [52], overloading, late binding [26], active rules [75], etc. It contains data modeling constructs as well as querying constructs. The following example illustrates the data definition constructs of AMOSQL by defining a type *person* and three stored functions over this type: *hobby* returning character strings, *name* returning a single character string, and *parent* returning *person* objects:

```
create type person;
create function hobby(person) ->  string as stored;
create function name(person)  ->  string key as stored;
create function parent(person) -> person as stored;
. . .
```

The keyword *key* limits the arity of a result or an argument to 0 or 1. The general syntax for AMOSQL queries is:

```
select <result>
   from <type declarations for local variables>
   where <condition>
```

The following example illustrates how functional views are defined with AMOSQL. Assuming the three stored functions *parent*, *name* and *hobby* from the example above, it defines a derived function that retrieves the names of those children of a persons having 'sailing' as a hobby:

```
create function sailing_children(person p) -> string as
  select n
    from person c
    where parent(c) = p and
          name(c)   = n and
          hobby(c) = 'sailing';
```

[1] The term "selectivity" is used throughout this thesis for the quantity commonly referred by both *selectivity* (when lower than 1) and *fan-out* (when greater than 1)

The query optimizer optimizes the function body and associates the produced query execution plan with the function. Since functions are used to represent properties of objects (i.e. methods) as e.g. *sailing_children*, the function bodies are always optimized assuming that the variables in the function arguments are *bound* while the other variables are initially unbound but will be assigned values when the function is executed. The term "bound" indicates that the variable has an assigned value before the execution of the function takes place. The result of an execution of a query is a subset of the unbound variables in the query. The variables which are neither in the result nor in the argument set of the query are named *local variables*. The local variables are unbound when a function execution begins. If, for example, the AMOSQL variable *:ip* represents a person instance, the expression:

```
sailing_children(:ip);
```

Invokes the function body with the variable `p` bound and the result variable n unbound. Alternatively, the query:

```
select p
  where sailing_children(p) = ''Tore'';
```

invokes the same function with the variable n bound, and the variable p unbound. The query retrieves the parents having a child named Tore with hobby sailing.

The ad hoc queries in AMOSQL are treated as functions without arguments. For example, assume the following query that retrieves the names of the parents of all persons having 'sailing' as hobby:

```
select p, name(parent(p))
  from person p
  where hobby(p) = 'sailing';
```

AMOS*II* processes this query by generating an anonymous function with no arguments, `query()`, which is executed immediately and then discarded:

```
create function query()-> <person, string>
  as select p, name(parent(p))
     from person p
     where hobby(p) = 'sailing';
```

3.3 Query processing in AMOS*II*

Figure 3.2, presents an overview of the query processing in AMOS*II*. The first five steps, also called *query compilation* steps, translate the body of a function (query) expressed in AMOSQL to a query execution plan which is stored with the function. To illustrate the query compilation we use the ad hoc query above.

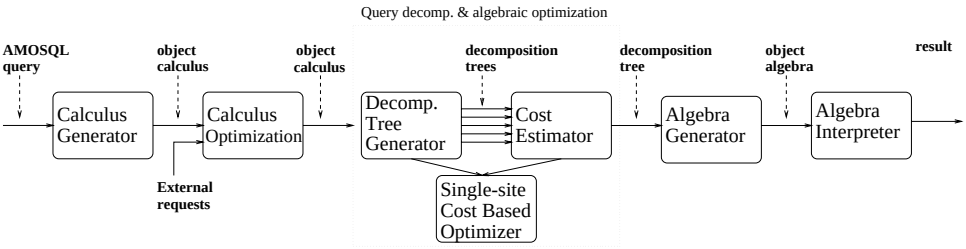


Figure 3.2: Query processing in AMOS*II*

From the parsed query tree, the calculus generator generates an *object calculus* expression. In the object calculus expressions, function symbols are annotated with *signatures* consisted of argument and the result types. Next, the calculus expression is transformed into a *flattened* form consisting of a set of equality predicates. The left-hand side of the equality predicates can be a single variable or a constant. It can also be a tuple of variables or constants when the right-hand side returns a tuple as a result. The right-hand side of a predicate can be an unnested function call, a variable, or a constant. The equality operator has semantics as in the DAPLEX query language where if the right hand side is multi-valued (bag), then the right hand side is compared (in case of a constant) or assigned (in case of a variable) to each of the values in the bag. The head of the calculus query expression contains the result variables. In the rest of the thesis, all calculus expressions will be shown in a flattened form. As an example, we consider the calculus representation of the ad hoc query above:

$$\{\ p, nm\ \mid$$
$$p = Person_{nil \rightarrow person}() \wedge$$
$$pa = parent_{person \rightarrow person}(p) \wedge$$
$$nm = name_{person \rightarrow string}(pa) \wedge$$
$$'sailing' = hobby_{person \rightarrow string}(p)\}$$

The first predicate in the expression is inserted by the system to assert the type of the variable p. It defines that the variable p is bound to one of the objects returned by the *extent function* of type $Person$, named $Person()$ and returns all the instances of this type. Besides being used to generate the extent of a type, the extent function can be also used to test if a given instance belongs to a type. Therefore, a predicate containing a reference to an extent function is called a *typecheck predicate*. An extent function accesses the *deep extent* of the type, i.e. it includes the extents of all the subtypes. By contrast, the *shallow extent function* considers only the immediate instances of the type. By convention, the shallow extent functions are named by prefixing the type name by the prefix *Shallow*, e.g. $ShallowPerson_{nil \rightarrow Person}()$.

AMOS*II* supports overriding and overloading of functions on the types of their arguments and results, i.e. their full signatures. Each function name refers to a *generic* function which can have several associated *type resolved* functions annotated with their signatures. During the calculus generation, each generic function call in a query is substituted by a type resolved one. Late binding is used for the calls which, due to polymorphism, cannot be resolved during query compilation [26].

Next, the calculus optimizer applies rewrite rules to reduce the number of predicates. In the example, it removes the type check predicate:

$$\{\ p, nm\ \mid$$
$$pa = parent_{person \rightarrow person}(p) \wedge$$
$$nm = name_{person \rightarrow string}(pa) \wedge$$
$$'sailing' = hobby_{person \rightarrow string}(p)\}$$

The type check predicate can be removed because p is used in a stored function (*parent* or *hobby*) with an argument or result of type $Person$. The referential integrity system of the stored functions constrains the instances of a stored function to the correct type [52]. If there is no such constraining function the query processor will retain type check predicates to guarantee that derived functions return correct result. For example, if the argument types of the functions *parent* and *hobby* had been supertypes of *person*, the

type check for p would have remained in the query to limit the processed instances to only the ones included in the *person* extent. As will be shown, the type check removal is particularly important for multi-database queries where type checks often need to cross database boundaries and are expensive.

Another rewrite rule used in this work is the *predicate unification* rule described in [23]. With this rule, two predicates with the same name and the same variables or constants for the key arguments can be combined into one. After the substitution, the non-key arguments of these predicates are pair-wise unified throughout the query. For example, in the following calculus expression:

$$\begin{array}{l}\{\ n1\ \mid \\ \quad n1 = name_{person\rightarrow charstring}(p)\ \wedge \\ \quad n2 = name_{person\rightarrow charstring}(p)\ \wedge \\ \quad foo_{charstring\rightarrow boolean}(n1)\ \wedge \\ \quad foo_{charstring\rightarrow boolean}(n2)\ \ \}\end{array}$$

the argument of the function $name_{person\rightarrow charstring}$ is a key and therefore the first two predicates can be replaced by one:

$$\begin{array}{l}\{\ n1\ \mid \\ \quad n1 = name_{person\rightarrow charstring}(p)\ \wedge \\ \quad foo_{charstring\rightarrow boolean}(n1)\ \wedge \\ \quad foo_{charstring\rightarrow boolean}(n1)\ \ \}\end{array}$$

This predicate is then further reduced to

$$\begin{array}{l}\{\ n1\ \mid \\ \quad n1 = name_{person\rightarrow charstring}(p)\ \wedge \\ \quad foo_{charstring\rightarrow boolean}(n1)\ \ \}\end{array}$$

The transformation is not correct if the transformed predicates have side-effects in the database or in the system's environment. The foreign functions in AMOS*II* are the only place in AMOS*II* where such side effects can be made[2]. Foreign functions that cause side-effects are tagged with a side-effect flag which will prevent application of this rewrite rule.

Because the example query is over local types, it passes unaffected

[2]Database procedures specified in AMOSQL extended with procedural constructs can have side effects too. In this case they are treated as foreign functions.

through the query decomposition stage and is processed only by the cost-based single-site algebra optimizer. If some part of the query is to be executed by another AMOS*II* server, the system will use primitives that allow for sending function definitions between the servers for local optimization and evaluation. The query decomposition will be discussed in detail in chapter 6.

The object calculus query representation is declarative and does not prescribe a certain evaluation order of the calculus predicates describing function calls. By contrast, the expressions in the object algebra [23] have a well defined evaluation order and are, in addition to the type annotations, annotated with *binding patterns* indicating which variables are input and which are output in each function call [52].

The calculus optimization process takes advantage of the declarative unordered format and the unspecified binding patterns of the object calculus for detection of optimization possibilities with the goal of reducing the number of query predicates by removing unnecessary computations. This optimization is rule-driven and much simpler than the transformations made during the cost-based algebraic optimization.

The query algebra used in AMOS*II* has six operators $\{\pi, \times, \cup, \cap, \bowtie, \gamma\}$, the first five of which have the same semantics as in the relational algebra. The last one, the γ operator, performs function application, and is similar to the generate operator of [77]. A formal definition of these operators can be found in [23]. Note that a selection operator is missing since it can be specified using a function application where some of the arguments are bound to constants, as shown in the next example.

Each type-resolved function in AMOS*II* can have several implementations with different binding patterns. Figure 3.3 shows two execution plans for the example query, expressed in the query algebra. In Figure 3.3a, a straight-forward translation of the query calculus expression to an algebra expression is given. The rectangles in this figure represent algebraic operators. The variables bound after each operator application are shown in-between the operators. The plan in figure 3.3a first applies the function *name*() over all the instances of type *person*; next, the children of a person are found by applying the function *parent*() "backwards" - giving a person as an input and retrieving its children; the third operator performs a selection based on the children's hobbies; and finally the required variables are projected from the selected tuples. The second plan, shown in Figure 3.3b, is more optimal. It first selects the persons with the required hobby, then

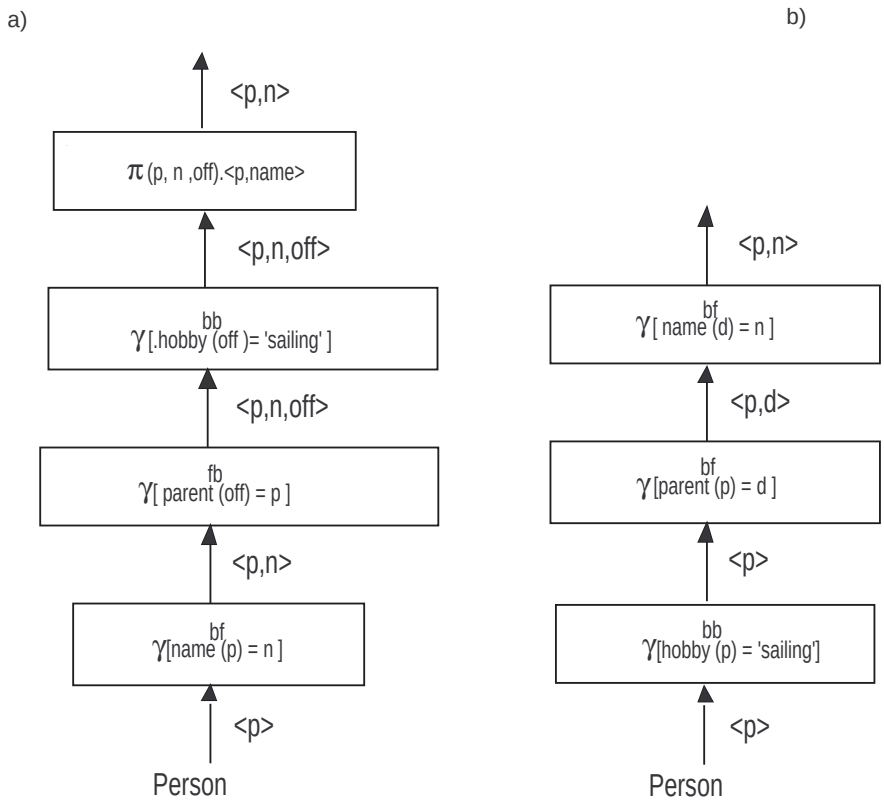


Figure 3.3: Two algebraic representations of the example query

finds their parents, and finally retrieves the parents' names. Note that each function is superscripted with the binding pattern used for its execution (although the functions are also type resolved, the type information is omitted for clarity). The vector representing the binding pattern has length equal to the added lengths of the function argument and result tuples. Each of the argument and result variables is associated with a flag in the binding pattern vector. In the figure, "f" is used for the *free* (unbound) variables and "b" is used for the bound variables. For example, the "fb" binding pattern for the function *parent*() means that the result is bound (a parent is given), while the argument is free (a child is returned). This kind of invo-

cation is related to the inverse function mechanism in some other models. Nevertheless, in AMOS*II* a function can be defined and executed using arbitrary binding patterns, generalizing thus the inverse function concept over functions with multiple arguments and results. Stored functions, as *parent*, can be efficiently executed with different binding patterns in the presence of matching secondary indices. Foreign functions can also have more than one implementation with different binding patterns and different user-supplied cost and selectivity functions [52].

The interested reader is referred to [27] for a more detailed description of the AMOS and AMOS*II* system and to [52, 23, 26] for more on the query processing in AMOS*II*. Previous work on data integration within the AMOS project is reported in [84].

Chapter 4

Data Integration by Derived Types

This chapter presents the basis of the Object-oriented (OO) view mechanism in AMOS*II*, used to provide the user with a unified appearance of data in different repositories. Queries over the views are transformed into queries over the data in the repositories. Passive data mediation, as described in this thesis, requires that the mediator system provides for complete and consistent answers to the queries over the OO views at the time when the queries are issued. An advantage of the passive approach described in this chapter is that it provides an efficient view support mechanism by describing the system tasks using predicates inserted in the calculus representation of the queries over the integrated views. This allows for query optimization of the view support tasks together with the user-specified part of the query. Another advantage is that the view maintenance operations, as well as the user-specified operations, are specified and performed over a set of objects/tuples as opposed to individual instances.

The focus of the chapter is a query transformation technique that, for a certain class of queries, allows for a reduction of the number of predicates by applying calculus-based optimization. The calculus-based optimization removes redundant computations that often result from merging system-specified and user-specified predicates in the query. This reduces the query complexity and, because it is performed by simple rewrite rules, it imposes a minimal increase in the query processing time. The cost-based optimization executed later in the query processing is concerned with the order of the

execution rather than the removal the redundant computations.

The rest of the chapter is organized in two sections. Section 2 introduces the OO views architecture for database mediation. Section 3 describes the query transformation techniques for the queries over the OO views and the use of rewrite rules to reduce the number of query predicates.

4.1 Object-oriented view system design

This section presents the design principles behind the OO view mechanism for data integration in AMOS*II*. Views as a tool for data abstraction and restructuring have been extensively studied in the context of the relational databases. The design of a view mechanism in an OO environment is more complex in particular with regards to inheritance and object identity. Inheritance and views have common aims (i.e. data abstraction and code reuse), and therefore the two mechanisms must be combined in a semantically clear manner. Two important issues in OO view system design are the format of the OIDs of the view objects and their life span. Additional issues for views defined over data in multiple data sources are non-intrusive mechanisms for view maintenance, managing semantic heterogeneity, and representation of OIDs in a distributed environment.

4.1.1 Derived types

To provide data integration features in AMOS*II*, the type system is extended with *derived types* (DTs) defined as subtypes of other types, and *integration union types* (IUTs), defined as supertypes of other types. Data integration by DTs and IUTs is performed by building an OO view type hierarchy based on local types, and types imported from other data sources, including other AMOS*II* servers. The traditional inheritance mechanism, where the corresponding instances of an object in the super/subtypes are identified by the same OID, is extended with declarative specification of the correspondence between the instances of the derived super/subtypes. Integration by sub/supertyping is related to the mechanisms in some other systems as, for example, the integrated views and column adding in the Pegasus system [17], but is better suited for use in an OO environment.

Figure 4.1 shows an example of using DTs for data integration by subtyping. In the example, the data stored in an employee database is integrated with data from a database containing sporting information. The solid ovals

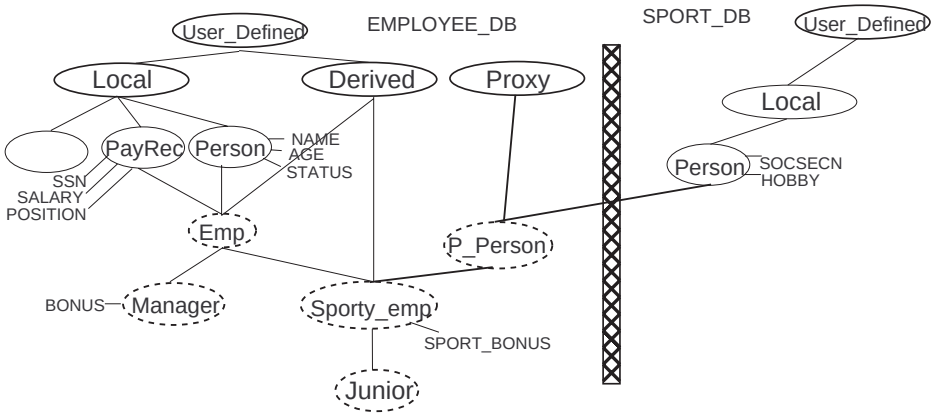


Figure 4.1: Integration by derived types (subtyping)

represent ordinary types while the dashed ovals are types created by the user and the system during the OO view definition process. Stored functions defined over the types in the figure are shown beside the type ovals. The types *User_Defined*, *Derived* and *Proxy* are system-defined and part of the meta-model in AMOS*II*. They are defined in both databases, but are not shown in *Sport_Database* for reasons of clarity. There is a type *Person* in both databases storing information about a set of persons. The definition of the derived portion of the type hierarchy is in the example done as follows. First, the DT *Emp* is created to represent the persons having a pay record. The DT *Manager* is a subtype of the DT *Emp* representing the employees for which the stored function *position* has the value 'Manager'. DTs can be used to integrate types in more than one data source by subtyping from types imported from other data sources. In the example, the DT *Sporty_Emp* is defined as a subtype of the local DT *Emp* and the type *Person* in the sport database. Its instances represent persons that are represented by an instance of both type *Emp* in the employee database, and type *Person* in the sport database.

The definition of the *Sporty_Emp* DT is stored in the type hierarchy of *Employee_DB*. *Sport_DB* stores no information about this type. To record that *Sporty_Emp* inherits from a type in another data source, the system au-

tomatically imports the type *Person* from *Sport_DB* into the *Employee_DB* database and defines a *proxy* type for it, named in the Figure *P_Proxy*. The proxy type mechanism is described in greater detail in the next section.

Figure 4.1 also illustrates some of our design choices. First, to be able to perform data integration by subtyping a multiple inheritance mechanism is required for the DTs. Second, it can be noticed in the example that stored functions (e.g. *sport_bonus* in *Sporty_Emp*) can be defined over DTs, which makes the DTs a *capacity-augmented view mechanism* [66]. DTs can be used in function definitions as ordinary types and any function can have DTs as argument or result domains.

4.1.2 Generation of OIDs for the DT instances

There are three basic choices for the format of OIDs representing DT instances. The first is to use the OIDs from the corresponding supertype objects [67]. This is not suitable in our case because it is not compatible with multiple inheritance. The second alternative is to use a stored query expression instead of an OID and construct the required DT instances by evaluating this expression [43]. With this approach, it would be difficult to have functions whose argument domain is a DT since it is not convenient to manipulate expressions as database objects. The third alternative, is to generate new unique OIDs for the DT instances [66]. With this method, the same conceptual object (i.e. representing the same real world entity) is represented by different OIDs in different types. Therefore, to be able to evaluate inherited functions over the DT instances, their OIDs need to be mapped to the OIDs of the corresponding instances of the type over which a function was defined, by a process named *OID coercion* [1]. The cost of OID coercion is the main weakness of this approach. Nevertheless, we chose this approach for the following two reasons: First, the major cost of a query is in accessing the data sources and shipping data among the AMOS*II* servers, and not in the coercion. In AMOS*II*, the hash tables used in the coercion are stored in a main-memory database that makes the coercion inexpensive. Second, expressing the coercion by predicates permits some query optimization that further reduces the coercion cost, as described in the next section.

Although the generation of OIDs for the DT instances allows for using the DTs as domains for function arguments and results, most queries over DTs require only a few or no OIDs and it would be a severe performance

[1] In the text we use the terms "OID coercion" and "instance coercion" interchangeably.

impairment to generate OIDs for the entire extents of all the DTs in each query. The OID generation cost includes the creation of a new OID and the storage of the coercion information in internal tables. To minimize this cost, and to avoid unnecessary creation of OIDs, the query processor analyzes the query to find out which query variables represent instances that need to be assigned OIDs. OID generation predicates are added only for query variables in the query result or used as arguments of foreign functions. Other queries are transformed so no OID generation is needed, as shown below. The query performance is thus not degraded by the OID generation mechanism. In queries requiring DT OIDs, these are generated selectively for those instances satisfying the rest of the query predicates, thus generating OIDs for only parts of the DT extents in order to avoid unnecessary performance and storage overheads.

DT OIDs stored in local functions can be used in queries issued after their generation. Then the system has to assert that the instances they represent still comply with the declarative conditions stated in the DT definition, i.e. that they are still *valid*. Assuming non-active and autonomous data sources, the system has to add run-time checks in the queries to check the validity of those DT instances that are previously imported from external data sources and stored in local functions in the mediator. These validation checks must access the corresponding data sources to check the validity of the exported DT instances. If the query does not access imported DT OIDs stored locally, the instances are retrieved directly from the data sources and no validation is needed.

The validity of a DT instance depends on the existence and validity of the corresponding supertype instances whose OIDs are stored in the coercion tables. When a DT instance is validated, the validation condition is executed only over these instances. This definition of the validity of a DT instance based on a validation condition over a tuple of supertype OIDs is consistent with the OO structure of the database, and is efficient to implement.

An instance is present in the mediator until it is used in a query where it fails the validation test. A garbage collection of the DT instances can be implemented to periodically run the validation test, deleting the instances not satisfying the test.

4.1.3 Derived types and inheritance

An important issue in designing an OO view system is the placement of the DTs in the type hierarchy. The obvious approach would be to place the DTs in the same hierarchy as the ordinary types. However, mixing freely DTs and ordinary types in a type hierarchy can lead to semantically inconsistent hierarchies [45]. In order to provide the user with powerful modeling capabilities along with a semantically consistent inheritance hierarchy, the ordinary and derived types in AMOS*II* are placed in a single type hierarchy where it is not allowed to have an ordinary type as a subtype of a DT. This rule preserves the extent-subset semantics for all types in the hierarchy. If DTs were allowed to be supertypes of ordinary types, due to the declarative specification of the DTs, it would not be possible to guarantee that each instance of the ordinary subtype (created explicitly by the user) has a corresponding instance in its derived supertypes.

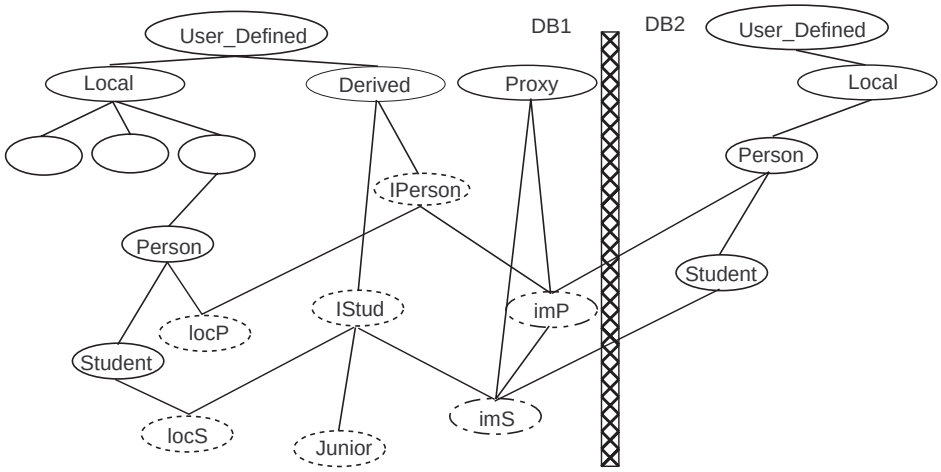


Figure 4.2: Integration by integration union types (supertyping)

In Figure 4.1 the view is constructed by subtyping. As noted earlier, the AMOS*II* integration framework also allows definition of declaratively defined IUTs as explicit supertypes of other types. Although the IUTs are described in detail in the next chapter, to complete the discussion on the

integration framework, Figure 4.2 presents an example of integration by IUTs. The example shows a definition of an integrated view of two person databases *DB1* and *DB2*. The data in both databases is structured in two user-defined types: a type named *Person* that contains data about a set of persons, and its subtype *Student* representing the persons who are students. The example establishes the IUT *IPerson* and *IStud* in DB1 to provide an integrated view of the data in the databases. These types represent the union of the real world entities represented by the instances of the integrated types in the both databases. In the example, the proxy types *imP* and *imS* represent the types *Person* and *Student* from *DB2*, imported into *DB1*. IUTs can also be subtyped by DTs. In this example the type *Junior* represents a specialization of the type *IStud* containing all junior students. The same schema was used in both databases in order to simplify the example. Using DTs, IUTs and derived functions, he presented integration framework can handle all schema heterogeneities that do not require higher-order language constructs.

4.1.4 Derived subtyping language constructs

For defining DTs as subtypes of other types, AMOSQL has the following construct:

```
CREATE DERIVED TYPE type_name
   SUBTYPE OF sut1, sut2, ...
      COMPOSE compose_expression
      VALIDATE validate_expression
   [ HIDE fn1, fn2, ... ]
   [ PROPERTIES (prop1 type_prop1, ....)] ;
```

The *subtype of* clause establishes the DT as a subtype of other types in the hierarchy. The *compose_expression* and *validate_expression* are boolean expressions which, when conjoined, make the condition that a combination of supertype instances needs to satisfy to compose a new DT object. The condition in *compose_expression* is evaluated only when an OID is generated for a new instance of a DT. By contrast, the condition specified with the *validate_expression* is also evaluated each time a query accesses OIDs of the DT stored locally in the mediator. The splitting of the composition and validation expressions was motivated by the observation that data integration is often performed on the basis of some key functions that do not change

over the lifetime of the instance (i.e. that are functionally dependent on the OIDs of the integrated instances). In these cases, it is not necessary to evaluate the full condition every time a DT instance is validated, but instead only the *validate_expression* is evaluated over the corresponding instances of the supertypes. Alternatively, in order to avoid the burden of this splitting, the user could specify the condition as one expression, and then it could be separated by the system into composition and validation expressions based on the key information of the stored and the foreign functions used in the expression. A drawback of this approach is that it cannot detect conditions over non-key function that do not change during the existence of an instance (e.g. that the age of a person has passed some limit). In the following example, defining three of the DTs in Figure 4.1, the condition expression is divided into the two parts:

```
create derived type Emp
  subtype of Person P, PayRecord PR
    compose ssn(P) = ssn(PR)
    validate  status(P) = 'working';

create derived type Sporty_Emp
  subtype of Person@SPORT_DB p, Emp e
    compose ssn(e) = adjust_ssn(socsecn(p));

create derived type Junior
  subtype of Sporty_Emp se
    validate age(se) > 26;
```

The function *adjust_ssn* converts a social security number stored in $SPORT\_DB$ to the format used in $EMPLOYEE\_DB$. This can be any kind of function defined locally or in $SPORT\_DB$, over strings and returning integers.

There is one instance of type *Emp* for each person having a pay record and status 'working'. Since the social security number does not change during the existence of a *Person*, the conditions involving the functions *ssn* and *socsecn* are in the *compose* clause of the definitions. On the other hand, the status and the age of a person can change and therefore the conditions over these functions are placed in the *validate* clauses.

The clauses *hide* and *properties*, which for brevity were not used in the examples, serve to list the functions of the supertypes not inherited by the

DT, and to define new stored functions, respectively.

4.2 Querying derived types

DTs differ from ordinary types in a number of ways. First, the extents of DTs are not stored in the database as the extents of ordinary types, but are defined by declarative functions. Next, if a function inherited by a DT is called in a query, the system needs to coerce the argument DT OIDs to the corresponding OIDs of the supertype where the function is defined. Here, although the instances have different OIDs, they correspond to the same conceptual object. Finally, the system must check the validity of the DT OIDs stored in local functions, when used after their creation.

These differences make the queries over DTs more complex and time-consuming than the queries over ordinary types. Naive evaluation of queries over the DTs, where the DTs are treated in the same way as the ordinary AMOS*II* types, leads to a very inefficient query evaluation strategy. It would first retrieve the extents of the DTs in the query, generate OIDs for them, and then apply the selection condition of the query. Arguments to function calls used in the query must be coerced correctly.

An analysis of the execution plans showed that most of the overhead can be avoided by introducing query transformations to:

- Avoid unnecessary OID generation.
- Reduce the coercion to a minimum.
- Allow for early application of selections in order to process only portions of the DT extents.
- Reuse OIDs stored in local functions instead of regenerating DT extents.

In order to achieve these goals, the OO views definitions are translated into system-defined derived functions. The calculus generator analyzes the query and, if the query is specified over DTs, inserts calls to these functions into the calculus representation of the query. Many OO view-support tasks traverse the type hierarchy and have common subtasks. The predicate representation of the derived function bodies allows these common subtasks to be identified and eliminated from the query together with overlaps between

user-defined and system-inserted predicates. Of particular interest in a view mechanism for data integration is to minimize operations that cross database boundaries in communication with other databases, or that access external data sources. Furthermore, the predicate-based view support approach allows selections from different query parts, such as user-specified and DT subtyping conditions, to be unified, optimized together, and applied as close as possible to the data sources. When a data source supports selection application (e.g. relational databases), the selections can be applied in the data source itself [23].

Although the common subexpression elimination mechanism allows for substantial reductions of queries over DTs, this alone does not remove all the redundancies in the queries. Therefore, two additional DT specific transformations are introduced to further eliminate redundant computation: First, queries over DTs having all functions inherited from their supertypes are transformed into queries over their supertypes. This eliminates all OID generation and coercion, as will be shown. Second, for queries accessing locally stored functions over DTs the system tries to reuse the locally stored DT OIDs instead of naively regenerating the DT extent again. The presence of a locally stored function limits the instances of interest to those stored in the function. However, since the OIDs stored in the local function were generated in previous transactions, and because of the autonomy of the data sources, the system needs to make sure that these OIDs still represent valid DT instances satisfying the validation condition of the DT. This validation could be avoided in some cases by, for example, the distributed query invalidation mechanism of [33].

In the rest of this section we will first describe how the DTs are modeled by AMOS*II* types and derived functions. Then, the query transformations are described in detail using example queries entered in the *EMPLOYEE_DB* mediator, over the views defined in the previous section. The section concludes with an algorithm for the calculus transformations.

4.2.1 Overview of the derived types implementation

Each DT in AMOS*II* is implemented by an ordinary local type named *implementation type*. The system automatically generates stored *coercion functions* over the implementation types to represent the mappings between those DT instances assigned OIDs and the tuples of corresponding instances of the DT's direct supertypes. All coercion functions are defined by the generic

function *coerce*, overloaded on both its argument and result. Coercion between an instance of a DT and its indirect supertypes is done by composition of coercion functions. The coercion functions are not accessible by the user. They are maintained by the system and used in system-defined functions generated from the DT definitions. For each DT the system generates three such functions: An *extent function*, a *validation function*, and an *OID generation function*. Informally, the extent function contains the subtyping condition and a call to the OID generation function. If invoked naively, it would generate all the tuples of the supertype objects that compose an object of the DT, and then invoke the OID generation function over these tuples to obtain OIDs for the DT instances. The OID generation function returns an already generated OID for each particular tuple of supertype instances, if such exists; otherwise, it creates a new OID and stores it in the coercion functions together with the tuple of supertype OIDs. Unlike the extent function, that contains the entire subtyping condition, the validation function contains only the DT validation condition. The validation function is used to check if a DT OID still represents a valid instance when used after its creation. The rest of this section presents the concepts named above in greater detail. To introduce the DT implementation, first the proxy mechanism for representation of data and types stored outside the mediator is presented.

4.2.2 Proxy types and objects

When a type from another data source is used for the first time as a supertype of a local DT, then it is either imported implicitly by the system (when an AMOS*II* data source is used), or explicitly by the *IMPORT TYPE* clause. Locally, for each imported type (distinguished by the type and data source name) a proxy type is created. All proxy types are subtypes of the type *Proxy*. For example, there is a proxy type, *P_Person*, defined for the type *Person* from the sport database. Figure 4.3 shows the proxy type hierarchy for the definition of the DT *Sporty_Emp* type in Figure 4.1. The proxy type hierarchy is first divided into different *kinds* of data sources. In the Figure, two kinds are shown: AMOS*II* data sources and ODBC data sources. This classification is not to be confused with the data source capability hierarchy described in chapter 6. Each data source is represented by a type placed under the type corresponding to the data source kind. There can be more than one data source of the same kind. Types imported from a data sources are placed in the type hierarchy under the type representing the source. Non-

OO data sources have flat proxy hierarchies (each proxy type is a child of the type representing the data source kind). The proxy types of OO data sources that can provide the required meta-data information are organized in a type hierarchy that is a subset of the type hierarchy in the exporting mediator, and contain all the types imported from the source to date. Locally defined DTs that are subtypes of types in other data sources are placed as subtypes of the corresponding proxy types.

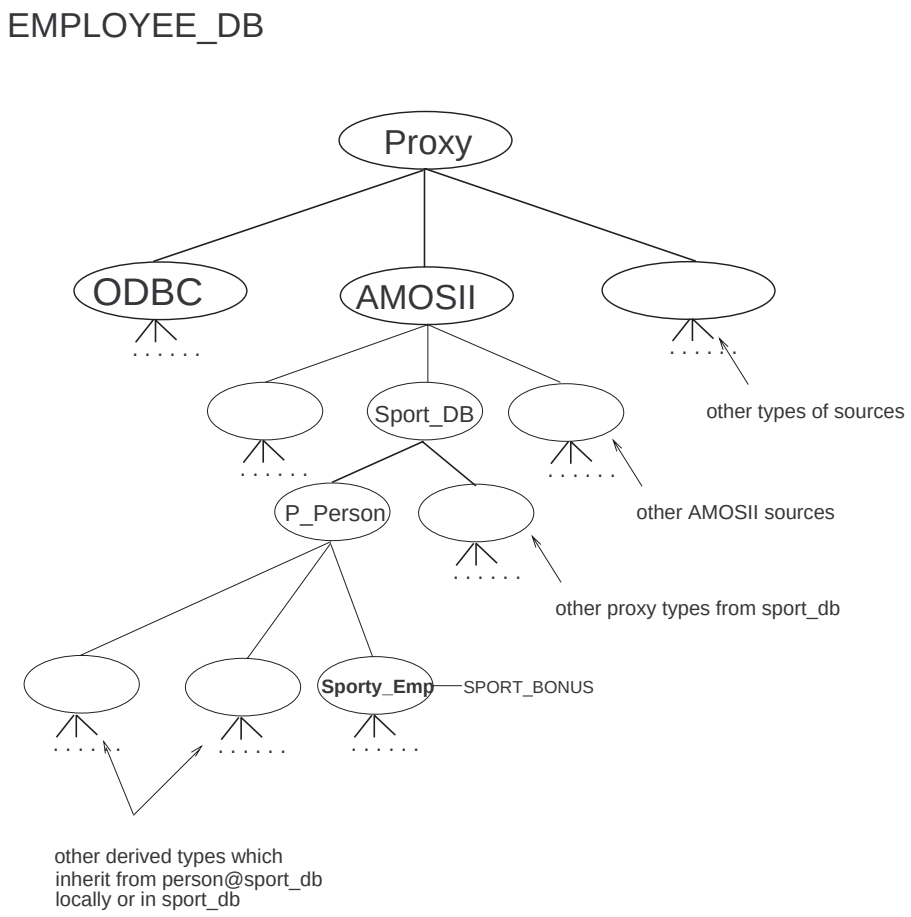


Figure 4.3: Placing the proxy types in the type hierarchy

After defining a proxy type, the system retrieves the signatures of the functions defined over the type in the exporting AMOS*II* server. If the argument and the result types of a remote function are known to the importing mediator (i.e. if they are system-defined types or previously imported user-defined types) a local corresponding *proxy function* is defined. The proxy function has the same signature as the remote function, but an empty body. Although the proxy functions and the proxy type extent functions are treated as ordinary functions throughout the calculus oriented query processing steps, they are not executed as ordinary functions. The decomposition algorithm groups them, and schedules them for execution in other AMOS*II* servers. In the calculus-based query processing phases, they provide information for type checking and query transformation as described below.

For each proxy type, a system-defined stored function is generated that maps instances of the proxy type into instances of type *foreign_oid*. This system type is used to represent the stringified OIDs received from other AMOS*II* servers when parts of query plans are evaluated there. The OIDs are transmitted among the mediators and stored in their native format without origin or typing information added. The OIDs generated by an AMOS*II* server are unique only within that server. The system makes no effort to generate "universal OIDs" unique in all AMOS*II* servers, like, for example, in the CORBA architecture [58]. In a CORBA environment, OIDs represent services and are designed to be transmitted alone. Therefore, every OID contains all the information needed to identify its origin. In a bulk data processing environment such as ours, the OIDs are passed in large collections having few different types and a common origin. Consequently, it is advantageous to condense the meta-information about the structure (types) and the origin of the transmitted OIDs with the transmission protocol. When an AMOS*II* server receives OIDs from another server, it stores them in their native format, while the meta-information is captured in the server's schema and the functions generated from the DT definitions. As a result of this kind of architecture, imported OIDs are stored in the mediator server, but they cannot be interpreted there. The user does not have direct access to the imported OIDs, but only to their proxy type instances. The system uses the imported OIDs only in operations executed in the server where they originate from. The main benefits from this approach are a simpler OID generation method, lower communication cost, and lower storage overhead due to smaller OIDs.

The name service in AMOS*II* provides means for a mediator to locate

other AMOS*II* servers that contain the types to be imported. AMOS*II* also provides an interface for providing information about the types to be imported by other mediators. This however, is not possible when types are imported from other types of data sources[2]. For this purpose, AMOSQL is expanded with constructs for data source declaration and explicit type importation:

```
IMPORT TYPE type_name@data_source
  [KEYS (key_list)]
  [FUNCTIONS (function_list)];
```

The KEYS clause defines a set of functions to be imported and used in the generation of OIDs for the instances of the proxy types representing data coming from non-OO sources. The FUNCTIONS clause can be used to import additional functions. The IMPORT TYPE clause can also be used to import types from AMOS*II* servers, when the user prefers to explicitly name the functions to be imported. If we assume that *Sport_DB* is an *ODBC* data source, then the data source declaration and the importation of the type *Person* would be specified as following:

```
DECLARE odbc DATA SOURCE Sport_DB;

IMPORT TYPE Person@Sport_DB
  KEYS (ssn integer)
  FUNCTIONS (hobby string);
```

In a query retrieving instances of the type $Person@Sport\_DB$, the generated calculus will instead use the proxy type P_Person. When OIDs are to be retrieved for instances of types imported from non-OO data sources, the wrapper amends the query so that the key functions (attributes) are retrieved instead. These are then used in the generation of the proxy instance OIDs, in a manner similar to the usage of stringified OIDs for proxy OID generation as described above. For more detail the reader is referred to [23].

4.2.3 DT extent function and template

The extent function of a DT is a system-generated derived function. The general form of the extent function is:

[2]While some types data of sources (e.g. databases) can provide the necessary information for automatic type importation, there are data sources that do not have this capability, making these language constructs necessary.

```
CREATE FUNCTION dt() -> dt AS
  SELECT genOID(s1, s2, ..., sn)
  FROM sut1 s1 , sut2 s2 ... sutn sn
  WHERE dt_compose_expression(s1, s2, ..., sn) AND
        dt_validate_expression(s1, s2, ..., sn);
```

where "dt" is the name of the DT, *sut1* ... *sutn* are the supertypes from the *subtype of* clause, and $genOID_{<sut1,sut2,...sutn>\rightarrow dt}$ is the OID generation function for the DT. *Dt_compose_expression* and *dt_validate_expression* are copied from the DT definition. If we represent these expressions as unexpanded derived functions, the calculus form of the body of the extent function would be:

$$
\begin{aligned}
&\{\ r\ \mid \\
&\quad s1 = sut1_{nil\rightarrow sut1}() \wedge \\
&\quad s2 = sut2_{nil\rightarrow sut2}() \wedge \\
&\quad \ldots \\
&\quad dt\_compose\_expression_{sut1,sut2...sutn\rightarrow boolean}(s1, s2, s3, \ldots, sn) \wedge \\
&\quad dt\_validate\_expression_{sut1,sut2...sutn\rightarrow boolean}(s1, s2, s3, \ldots, sn) \wedge \\
&\quad r = genOID_{sut1,sut2...sutn\rightarrow dt}(s1, s2, s3, \ldots, sn)\}
\end{aligned}
$$

Now we consider the problem of calculating the result of a function inherited by a DT. To illustrate the steps needed for this we use the DT *Emp* and the function $name_{person\rightarrow string}$ from the example above, although the same principles apply for any DT and any inherited function. The query on the left below retrieves the names of all the employees; the calculus generated for this query is given on the right:

```
select name(se)
from Emp se;
```

$$
\begin{aligned}
&\{\ n\ \mid \\
&\quad e = Emp() \wedge \\
&\quad p = coerce_{emp\rightarrow person}(e) \\
&\quad n = name_{person\rightarrow string}(p)\}
\end{aligned}
$$

The extent function *Emp*() produces the instances of the DT *Emp*. The stored function *name* stores OIDs of type *Person*. Since the instances of the DT *Emp* have OIDs different from the OIDs of the corresponding instances in the DT *Person*, they need to be coerced before applying the function *name* defined over *Person* instances. Expanding the *Emp()* extent function produces the following:

$$
\begin{aligned}
&\{\ n \mid \\
&\quad p = Person_{nil \to person}() \wedge \\
&\quad pr = PayRec_{nil \to payrec}() \wedge \\
&\quad emp\_compose\_expression_{<person,payrec> \to boolean}(p, pr) \wedge \\
&\quad emp\_validate\_expression_{<person,payrec> \to boolean}(p, pr) \wedge \\
&\quad p = coerce_{emp \to person}(e) \wedge \\
&\quad n = name_{person \to string}(p) \wedge \\
&\quad e = genOID_{<person,payrec> \to emp}(p, pr)\}
\end{aligned}
$$

Notice that this query can be simplified by removing calls to the OID generation and coercion functions since the variable e is not used in the result.

In this simple example it is easy to spot and remove the unnecessary predicates. In a more elaborate example with several nested DT extent and coercion functions it would be difficult to perform these removals. Therefore, for this type of optimization we have developed an approach in which the optimized query is generated by a set of transformations from the initial query calculus representation. During these transformations, instead of a complete extent function, an *extent template* (ET) is used. For each DT, an ET is generated from the calculus representation of the extent function. ETs have signatures and bodies. The signature contains a name, a list of *substitute variables* (SVs), and list of types associated with the SVs. The SVs are the variables used as arguments of the OID generation function in the extent function ($s1 \ldots sn$ in the general form of the extent function above). There is one SV for each supertype of the DT. The body is a predicate template consisting of the extent function body without the OID generation predicate.

The term 'template' is used instead of 'function' because the ETs do not satisfy all the formal requirements to be classified as functions. Templates are used only for function transformations and have only calculus representations that cannot be executed. Also, the template expansion rules differ from the rules used for function expansion. The following example shows the ETs for the DTs *Sporty_Emp* and*Junior* and *Emp* in Figure 4.1:

signature:
$ET\_sporty\_emp_{<P\_Person,emp>} : \_px, \_e$
body:
$\_px = P\_Person_{nil \rightarrow P\_Person}() \wedge$
$\_e = ET\_emp_{<person,payrec>} \wedge$
$sssn = socsec_{P\_Person \rightarrow string}(\_px) \wedge$
$essn = ssn_{person \rightarrow int}(\_e) \wedge$
$essn = adjust\_ssn_{string \rightarrow int}(sssn)$

signature:
$ET\_junior_{sporty\_emp} : \_se$
body:
$\_se = ET\_sporty\_emp_{<P\_Person,emp>} \wedge$
$a1 = age_{person \rightarrow int}(\_se) \wedge$
$26 > a1$

signature:
$ET\_emp_{<person,payrec>} : \_p, \_pr$
body:
$assn = ssn_{payrec \rightarrow int}(\_pr) \wedge$
$assn = ssn_{person \rightarrow int}(\_p) \wedge$
$'working' = status_{person \rightarrow string}(\_p)$
By convention, ET names begin with the *ET* prefix. Each template name is subscripted with the SV types, while the SVs are listed after the colon. An expression with a variable as the left-hand side and an ET as a right-hand side is named an *ET declaration*. An ET declaration is added to the query for each variable declared with a DT. It asserts the type of a DT variable, analogous to the extent function of the ordinary types. When a DT is defined by subtyping from other DTs, its ET body can contain nested ET declarations, as for *ET_sporty_emp* and *ET_junior* above.

The ET body contains predicates to assert that a tuple of instances of the supertypes composes an instance of the DT. Because the ETs are not complete functions, a calculus expression containing ETs is considered *incomplete*. In the calculus generation phase, the incomplete calculus expression containing ET declarations is transformed to a complete calculus expression by a series of transformations performed until there are no more ET declarations. In such a transformation, an ET declaration of a variable is removed from the query if the variable can be type checked by being used

as a function argument of the same DT. Otherwise, *ET expansion* is performed. During ET expansion, first the ET declaration is substituted by the ET body. Then, each occurrence of the variable declared by the ET declaration is substituted in *the rest of the query predicates* by a SV in the ET signature having the same type or a supertype of the argument's type. An ET expansion transforms a query over a DT into a query over its supertypes, thus avoiding OID generation and run-time coercion. Note that this kind of variable substitution differs from the substitution in normal function expansion where the argument and result variables in the function body are substituted to match the parameters.

The ET expansion process is illustrated through the example query below on the left over the schema in Figure 4.1. It is first translated to an incomplete calculus expression given below on the right:

```
select salary(j), age(j)
from Junior j
where hobby(j)='golf';
```

$$\{\ sal, a\ \mid\ j = ET\_junior_{Sporty\_Emp}\ \wedge\ sal = salary_{payrec\rightarrow int}(j)\ \wedge\ a = age_{person\rightarrow int}(j)\ \wedge\ 'golf' = hobby_{P\_Person\rightarrow string}(j)\}$$

The ET declaration of the variable j is not removed because j is not used as an argument or result of type *Junior* in any function in the query. Therefore, this ET is expanded and all occurrences of j in the query body are substituted by the template variable $\_se$ in ET_sporty_emp. The expression produced by this expansion (the first expression below) contains an ET declaration ET_sporty_emp. Analogous to the variable j ET declaration, this ET is also expanded yielding the second expression below:

$$\{\ sal, a\ \mid\ \_se = ET\_sporty\_emp_{<P\_Person,emp>}\ \wedge\ a1 = age_{person\rightarrow int}(\_se)\ \wedge\ 26 > a1\ \wedge\ sal = salary_{payrec\rightarrow int}(\_se)\ \wedge\ a = age_{person\rightarrow int}(\_se)\ \wedge\ 'golf' = hobby_{P\_Person\rightarrow string}(\_se)\} \quad (1)$$

$$
\begin{aligned}
&\{\ sal, a\ \mid && (2)\\
&\ \_px = P\_Person_{nil \rightarrow P\_Person}() \wedge\\
&\ \_e = ET\_emp_{<person,payrec>} \wedge\\
&\ sssn = socsec_{P\_Person \rightarrow string}(\_px) \wedge\\
&\ essn = ssn_{person \rightarrow int}(\_e) \wedge\\
&\ essn = adjust\_ssn_{string \rightarrow int}(sssn) \wedge\\
&\ a1 = age_{person \rightarrow int}(\_e) \wedge\\
&\ 26 > a1 \wedge\\
&\ sal = salary_{payrec \rightarrow int}(\_e) \wedge\\
&\ a = age_{person \rightarrow int}(\_e) \wedge\\
&\ 'golf' = hobby_{P\_Person \rightarrow string}(\_px)\}
\end{aligned}
$$

In the *salary* and *age* functions, the variable *_se* of type *Sporty_Emp* is substituted by the SV *_e* of type *Emp* through which these functions are inherited in *Sporty_Emp*. By contrast, in the *hobby* function, *_se* is substituted by the variable *_px* since this function is inherited through the *P_Person* type.

Finally, the ET declaration of the the variable *_e* is expanded. After this expansion the query expression does not contain any ET declarations:

$$
\begin{aligned}
&\{\ sal, a\ \mid\\
&\ \_px = P\_Person_{nil \rightarrow P\_Person}() \wedge && (*)\\
&\ assn = ssn_{person \rightarrow int}(\_p) \wedge && (2)\\
&\ assn = ssn_{payrec \rightarrow int}(\_pr) \wedge\\
&\ 'working' = status_{person \rightarrow string}(\_p) \wedge\\
&\ sal = salary_{payrec \rightarrow int}(\_pr) \wedge\\
&\ sssn = socsec_{P\_Person \rightarrow string}(\_px) \wedge && (*)\\
&\ essn = adjust\_ssn_{string \rightarrow int}(sssn) \wedge\\
&\ essn = ssn_{person \rightarrow int}(\_p) \wedge && (2)\\
&\ a1 = age_{person \rightarrow int}(\_p) \wedge && (1)\\
&\ 26 > a1 \wedge\\
&\ a = age_{person \rightarrow int}(\_p) \wedge && (1)\\
&\ 'golf' = hobby_{P\_Person \rightarrow string}(\_px)\} && (*)
\end{aligned}
$$

The first nine predicates are results of ET declaration expansions. The last three predicates originate in the original query. The calculus optimizer further reduces the example expression by unifying pair-wise the predicates indicated by the same number on the far right (the re-write rule is described in [23]). In case (1) there is an overlap between the user-specified query pred-

icates and the validation expression of DT *Junior*. In case (2) the definitions of the DTs *Sporty_Emp* and *Emp* overlap. The query calculus expression now contains six system-inserted predicates. The result of the query optimization is then processed by the query decomposition algorithm that, in this example, combines the three predicates marked with (*) for execution in the sport database. There, the local optimizer will further remove the type check predicate (the first predicate) since it has the information needed to deduce its redundancy. The queries produced by the decomposer in the mediator and sent to the two servers are:

in EMPLOYEE_DB

$\{\ sal, a, sssn\ \mid$
$assn = adjust\_ssn_{string \rightarrow int}(sssn) \wedge$
$assn = ssn_{person \rightarrow int}(\_p) \wedge$
$assn = ssn_{payrec \rightarrow int}(\_pr) \wedge$
$'working' = status_{person \rightarrow string}(\_p) \wedge$
$sal = salary_{payrec \rightarrow int}(\_pr) \wedge$
$a = age_{person \rightarrow int}(\_p)\ \wedge 26 > a\}$

in SPORT_DB

$\{\ sssn\ \mid$
$sssn = socsec_{Person \rightarrow string}(\_px) \wedge$
$'golf' = hobby_{Person \rightarrow string}(\_px)\}$

The queries are executed in each of the servers and then an equi-join over *sssn* is performed in the site determined by the query decomposer, based on the costs of execution and data transfer. The only data transferred between the servers will be the set of social security numbers of the relevant persons, thereby avoiding generation of OIDs for the queried types.

The transformations of the extent templates shown above reduce the need for run-time coercing. In this example, where the query does not return OIDs and is not evaluated over local functions storing DT OIDs, no coercion or OID generation predicates are needed in the final query. By modeling the extent generation by predicates these predicates are unified with user specified selections that further reduce the processing.

4.2.4 Generation of OIDs for DT instances

The preceding subsection demonstrated calculus generation and optimization where the generation of OIDs for the DT instances can be avoided altogether. This subsection briefly describes how the DT instances are assigned OIDs in queries requiring this. Let's consider the following *set* command:

```
set sport_bonus(e) = 1000 from emp e where salary(emp) > 1000;
```

Here, the system first retrieves OIDs of type *Emp*, and then stores them with the bonus in the locally stored function *sport_bonus*. The generation of DT OIDs in this update query cannot be avoided.

An OID is generated for a DT instance if it is a part of the query result or used as an argument to a foreign function. OID generation functions are implemented as system-generated foreign functions taking as argument a tuple of DT supertype OIDs and returning a DT OID. If for the given arguments there is an already generated OID, it is returned without creating a new one. The OID generation functions are defined by the system as resolvents of the overloaded function *genOID*.

When, a calculus variable ranges over DT instances assigned an OID, the extent template defining this variable is replaced with the expanded extent function. The following example illustrates this process. The query on the left returns an instance of the DT *Manager*. The expanded object calculus generated for this query (shown on the right) contains two OID generation predicates. When an OID for a DT instance is generated, the OIDs of the corresponding instances in the derived supertypes need to be generated too. Therefore, in the example, the system also inserts an OID generation predicate for the DT *Emp*.

```
select m into :john
from manager m
where name(m) = 'John'
```

$$
\begin{aligned}
&\{\ m \mid \\
&\quad s = ssn_{person \rightarrow int}(p) \wedge \\
&\quad s = ssn_{payrec \rightarrow int}(pr) \wedge \\
&\quad 'John' = name_{person \rightarrow string}(p) \wedge \\
&\quad 'Manager' = position_{payrec \rightarrow string}(pr) \wedge \\
&\quad e = genOID_{<person,payrec> \rightarrow emp}(p, pr) \wedge \\
&\quad m = genOID_{emp \rightarrow manager}(e)\ \}
\end{aligned}
$$

The *into* clause stores the query result into an AMOSQL variable.

To limit the OID generation to only the requested DT instances, the OID generation predicates should appear late in the final query execution plan after query conditions restricting the number of generated OIDs. The

optimizer is aware of this, and after performing the cost-based optimization it moves the OID generating expressions to the end of the query execution plan, preserving their relative order. Because the displaced expressions have low cost and selectivity 1, this transformation does not affect the overall query cost. This strategy is applicable to queries where OIDs are generated for DT instances in the query result, as in the example above. When the generated OIDs are used in some foreign functions, more elaborate interactions between the calculus generator and the algebra generator are required. This mechanism is not described in this thesis.

4.2.5 Processing of queries using locally stored functions

As shown above, instances of a DT from a data source can be assigned OIDs and stored in local functions over the DT. These stored functions can be later referenced in user queries. Then, because the data in the data source can change without the control of the mediator, DT OIDs retrieved from the locally stored functions need to be validated. Note, however, that no action is needed when new instances are added in the data sources, since these new instances must be first stored in a local function in the mediator before any validation is needed. For example, if a person takes up golfing and thus becomes a *Sporty_Emp*, this person's OID need not be validated until it is stored in a local function. Furthermore, the fact that the locally stored functions are cheap to access, and most often store only portions of the DT extent, can be used by the optimizer to produce plans operating only over the DT instances stored in these functions instead of the entire DT extent.

To illustrate the processing of queries with locally stored functions over DTs, we extend the example from section 3.4.2 with a predicate (underlined) over the locally stored function *sport_bonus*, defined over the instances of the DT *Sporty_Emp*:

```
select age(j), salary(j)
from Junior j
where hobby(j)='golf' and
      sport_bonus(j) > 100;
```

$$
\begin{aligned}
&\{\, a, sal \mid \\
&\quad j = ET\_junior_{Sporty\_Emp} \wedge \\
&\quad \underline{b = sport\_bonus_{sporty\_emp \rightarrow int}(j) \ \wedge\ b > 100 \wedge} \\
&\quad \underline{a = age_{person \rightarrow int}(j) \wedge} \\
&\quad sal = salary_{payrec \rightarrow int}(j) \wedge \\
&\quad 'golf' = hobby_{P\_Person \rightarrow string}(j)\}
\end{aligned}
$$

As in the previous example, first a reference to *ET_junior* is inserted and expanded. The resulting query contains an ET declaration of the variable *_se*

with *ET_sporty_emp*. Furthermore, the variable j is substituted by the variable *_se* throughout the query. At this point, since the variable *_se* is used as an argument of the function $sport\_bonus_{sporty\_emp \rightarrow int}$, *ET_sporty_emp* is not expanded, but instead removed. The variable *_se* in this case iterates only over the already materialized portion of the extent of *Sporty_Emp*, stored in $sport\_bonus_{sporty\_emp \rightarrow int}$.

For a correct expression, the transformed query expression needs to be extended with predicates to perform the coercion and validation of the instance OIDs of *Sporty_Emp*. This can be described as:

$$
\begin{array}{ll}
\{\, a, sal \mid & \\
\quad b = sport\_bonus_{sporty\_emp \rightarrow int}(\_se) \ \wedge\ b > 100 \ \wedge & \\
\quad \textbf{validate \_se} \ \wedge & (1) \\
\quad \textbf{coerce \_se to p of person} \ \wedge & (2) \\
\quad a = age_{person \rightarrow int}(p) \ \wedge & \\
\quad a1 = age_{person \rightarrow int}(p) \ \wedge\ 26 > a1 \ \wedge & \\
\quad \textbf{coerce \_se to pr of payrec} \ \wedge & (3) \\
\quad sal = salary_{payrec \rightarrow int}(pr) \ \wedge & \\
\quad \textbf{coerce \_se to px of P\_Person} \ \wedge & (4) \\
\quad 'golf' = hobby_{P\_Person \rightarrow string}(px)\} &
\end{array}
$$

The lines in bold give abstract descriptions of the operations added by the system. The numbers on the far right are for reference purposes. The predicates containing the variable $a1$ are inserted when the ET of type *Junior* is expanded.

The validation function ensures that the corresponding instances of the supertypes are still present and valid in the data sources, and that the validation condition evaluated over these instances still holds. Its general form is:

```
CREATE FUNCTION validate_DT(DT obj) -> boolean AS
  SELECT TRUE
  FROM sut1 st1, sut2 st2, ...
  WHERE st1 = coerce(obj) AND
        validate_st1(st1) AND
        st2 = coerce(obj) AND
        validate_st2(st2) AND ...
        validate_predicate;
```

The function coerces the argument to each of the corresponding super-

type instances, validates these instances, and then evaluates the validation condition. For example, the validation function for the DT *Emp* in Figure 4.1 is as follows:

```
CREATE FUNCTION validate_emp(emp e) -> boolean
   SELECT TRUE
   FROM Person p, Payrec pr
   WHERE p = coerce(e) AND status(e) = 'working' AND
         pr = coerce(e);
```

The validation function of a proxy type performs a check whether the corresponding $foreign\_OID$ instance exists in the database it originates from. This is implemented by a single type check predicate.

The coercion and validation in the example above require the following 11 predicates to be inserted into the query:

$$e = coerce_{sporty\_emp \to emp}(\_se) \ \wedge \ pi0 = coerce_{sporty\_emp \to P\_Person}(\_se) \wedge (1)$$
$$p = coerce_{emp \to person}(e) \ \wedge \ pr = coerce_{emp \to payrec}(e) \wedge$$
$$'working' = status_{person \to string}(p) \ \wedge pi0 = P\_Person_{nil \to P\_Person}() \wedge$$
$$e1 = coerce_{sporty\_emp \to emp}(\_se) \ \wedge \ p = coerce_{emp \to person}(e1) \wedge \quad (2)$$
$$e2 = coerce_{sporty\_emp \to emp}(\_se) \ \wedge \ pr = coerce_{emp \to payrec}(e2) \wedge \quad (3)$$
$$px = coerce_{sporty\_emp \to P\_Person}(\_se) \quad (4)$$

The numbers on the left match the predicate groups with the corresponding task in the previous query. After inserting these predicates in the query, the optimizer, by predicate unification and type check removal, reduces the number of system inserted predicates from 11 to 6. In addition, the query optimizer removes one of the calls to the *age* function. The resulting query is:

$$\{ \ a, sal \ \mid$$
$$b = sport\_bonus_{sporty\_emp \to int}(\_se) \ \wedge \ b > 100 \wedge$$
$$e = coerce_{sporty\_emp \to emp}(\_se) \wedge$$
$$p = coerce_{emp \to person}(e) \ \wedge \ 'working' = status_{person \to string}(p) \wedge$$
$$a = age_{person \to int}(p) \ \wedge \ 26 > a \wedge$$
$$pr = coerce_{emp \to payrec}(e) \ \wedge \ sal = salary_{payrec \to int}(pr) \wedge$$
$$px = coerce_{sporty\_emp \to P\_Person}(\_se) \wedge$$
$$px = P\_Person_{nil \to P\_Person}() \ \wedge \ 'golf' = hobby_{P\_Person \to string}(px) \}$$

The query decomposer will divide the query predicates into two functions: one executed in $EMPLOYEE\_DB$ and the other in $SPORT\_DB$. The

$EMPLOYEE\_DB$ function contains all the predicates except the last two. The function in $SPORT\_DB$ is compiled from the last two predicates and the typecheck is removed by the optimizer (the $EMPLOYEE\_DB$ function below is abbreviated for brevity):

in EMPLOYEE_DB

$\{ a, sal, px \mid$
$b = sport\_bonus_{sporty\_emp \rightarrow int}(\_se) \wedge$
$\ldots$
$px = coerce_{sporty\_emp \rightarrow P\_Person}(\_se)\}$

in SPORT_DB

$\{ px \mid$
$'golf' = hobby_{Person \rightarrow string}(px)\}$

Notice that in this case OIDs are shipped from one AMOS*II* server to another. Assuming that the function *sport_bonus* in $EMPLOYEE\_DB$ has a smaller extent than the function *hobby* in $SPORT\_DB$, the decomposer will generate a schedule in which the function on the left above is executed first and the stored OIDs are shipped to $SPORT\_DB$. There, the function on the right is executed, performing an equi-semi-join of the shipped OIDs with the function *hobby*.

4.2.6 The Transformation algorithm

We conclude the discussion of the DT query transformations with an algorithm for the described transformations. The input of the algorithm is a conjunction of predicates and a list of result variables. The output is a predicate in which all the DT extent functions have been transformed or expanded. The algorithm assumes that the input predicate is a conjunction of simple (non-derived) predicates and DT extent functions. Nevertheless, it can easily be expanded to predicates containing nested disjunctions and derived predicates. Also, single argument functions are assumed, to simplify the presentation.

The following functions are assumed to be predefined: $et\_body(dt)$ returns the body of the extent template of dt; $et\_sv(dt)$ returns the substitution variables from the signature of the extent template of dt; $type(var)$ returns the type of a calculus variable; $expand\_function$ substitutes a function call with its already expanded function body; the '$<$'and '$\leq$' operators represent subtype/supertype comparison; the $\cup$ operator is used for appending con-

junctions of predicates and adding a predicate to a conjunction.

```
expand_DT_extent_functions( P, resVars; ) -> PR
oidGen := resVars;
PR := P;
while ∃ J ∈ PR : J ≡ ( X = dt_{nil→dt}() ) ∧
                     dt() is extent func. of the DT dt
  REST := PR − dt_{nil→dt}();
  if X ∈ oidGen then do
     /* generate OIDs for the supertypes */
     oidGen := oidGen ∪ et_sv(dt);
     PR := expand_function(dt_{nil→dt}()) ∪ REST;
  else
    if ∃ J ∈ REST : J ≡ ( f_at(X) ) ∧ at ≤ dt ∧ f_at is stored func. then
       PR := expand_function(validate_at(X)) ∪ REST;
    else
      for each R ∈ REST
        if R ≡ ( Q_bt(X) ) then do    /* R is over the variable X */
          T := ( Q_bt(Y) ) ∧ Y ∈ et_sv(dt) ∧ type(Y) ≤ bt;
          PR := PR ∪ T
        else
          PR := PR ∪ R;
        end if
      end for each
      PR := PR ∪ et_body_{type→predicate}(dt)
    end if
  end if
end while
```

The **while** loop is executed until there are no more DT extent functions in the predicate. For a chosen DT extent function, the first **if** checks if the DT variable belongs to the set of variables representing instances that are to be assigned OIDs. If so, the DT extent function is substituted with its body and the variables representing instances of the supertypes are added to the list of types for which OIDs are generated. Else, if there is a predicate containing a locally stored function over the DT dt in PR, then the validation function is inserted and expanded; otherwise the predicate is traversed, all occurrences of the variable X are substituted with the supertype variables, and the template body is appended.

4.3 Database updates and coercing

In the polymorphic data model of AMOS*II*, a stored function defined over a type can store not only objects of that type, but also of all its subtypes. If instances returned by an evaluation of a stored function are used as arguments of another (consumer) function, they first need to be coerced. The coercion starts at the most specific type and ends in the type used in the consumer function argument declaration. Because of the polymorphism, the instances returned by the producer function can be of different most specific types, forcing the system to choose among different coercing sequences during runtime. This would require a complicated coercing expression that would degrade query performance. The following example illustrates this situation:

```
create function best_employee()->Emp e;
select m into :best_manager
  from manager m
  where bonus(m) = 1000000;
set best_employee() = :best_manager;
select name(best_employee());
```

In the example, first a function with no argument storing an instance of type *Emp* is created. Then, a manager is selected into the variable *:best_manager*. The *set* command sets the value of the function *best_employee()* to *:best_manager*. This operation is possible because the type *Manager* is a subtype of the type *Emp*. Now, when the name of the employee stored in *best_employee()* is requested, the coercion function needs to determine the most specific type of the stored instance (i.e. *Emp* or *Manager*) to be able to define the coercing process from that type to the type *Person* where the function *name* is defined.

To resolve this problem, AMOS*II* asserts that the most specific type of the stored instances is the same as the type specified in the function's definition. This is done by coercing the DTs' instances to the type in the function's definition when they are stored in a function. Assuming higher frequency of queries than updates, this enhances the performance of the system. In the example above, when the *set* command is executed, the instance stored in *:best_manager* is coerced to its corresponding *Emp* instance before it is stored.

Chapter 5

Integration of Overlapping Data

The data and the meta-data (schema) in the data sources can have conflicting and overlapping portions. For example, two universities can each have employee databases organized in different ways with corresponding entities bearing different names. Also, there might exist employees employed by both universities. The previous chapter described a framework for reconciliation of naming, scaling and other *object class heterogeneity*. This chapter will concentrate on a framework for mediating a coherent view of databases in the presence of *object instance heterogeneity*, where there is an overlap between the sets of real-world entities represented by the data in the sources.

In particular, this chapter deals with managing OO mediator views defined as unions of real-world entities from other AMOS*II* systems and data sources. Our mediating union views are modeled by a mechanism called *integration union types* (IUTs) based on OO queries and views. The IUTs model unions of real-world concepts similar to [14, 17], but opposed to unions of type extents from different databases as in [81, 36]. IUTs have *reconciliation* facilities that allow the user to specify how overlaps and conflicts between data from different sources are resolved.

Users and applications using a mediator often need to associate some locally relevant data to the data integrated from the data sources. We call such mediators, permitting local methods and attributes in the OO views, *capacity augmenting mediators*. Capacity augmentation for the IUTs is achieved by making the instances of the IUTs first-class objects with their own OIDs

that can be used in locally stored attributes and methods as ordinary OIDs.

The data sources are autonomous and can be updated outside the control of the mediators. The system must therefore guarantee the consistency and completeness of queries to the capacity augmented mediators in the presence of updates to the data sources. Our framework for IUTs guarantees that queries to the mediators are consistent and complete when the data sources are updated without any need for a notification mechanism. The queries over the integrated views always return *all* answers that meet the query condition, and *only* those answers that qualify, based on the *current* state of the data in the data source, regardless of any state materialized in the mediator.

It is challenging to achieve acceptable performance of OO queries over IUTs, in particular when the integrated extents have overlaps [14, 17]. Such overlaps require outer-join-based query processing techniques having increased complexity compared to inner joins. Furthermore, queries involving both local and remote data should take advantage of the fast access to local data to improve performance.

This chapter presents a combination of query processing strategies that significantly improve the performance of queries over IUTs in capacity augmented mediators. The main principles of these strategies are:

1. The IUTs are internally represented as a set of *auxiliary views*, over which the reconciliation is specified by a set of overloaded auxiliary methods (queries). This is supported by extending the overloading mechanism to cover declaratively defined OO views.

2. The queries over the IUTs containing outer-joins and reconciliation are translated into queries containing late bound calls of the auxiliary methods, over the auxiliary views.

3. In order to permit further query rewrites, the late bound queries are translated into disjunctive query expressions. These model the original query by joins and anti-semi-joins that are easier to rewrite and optimize.

4. Novel, type-aware query rewrite techniques remove inconsistent disjuncts and simplify the transformed disjunctive queries.

5. To efficiently support consistent and complete query answers the system uses a novel technique for selective OID generation and validation of the OO view instances, based on declarative queries.

6. Finally, local main-memory indexes created on-the-fly in mediators eliminate repeated accesses to data sources.

Experimental results show that the combination of the above methods has drastically better performance than a naive CORBA-like integration that resolves late binding on an object instance level at run time. The performance is drastically reduced even if only some of the combined optimization methods are relaxed.

The chapter is organized as follows. Section 5.1 describes the OO views framework and how is it used to model the user's view of the data in the repositories. Section 5.2 describes the system support for the ITs and the processing of the queries over the ITs. In section 5.3 some experimental results are presented and discussed.

5.1 Integration union types

The integration union types (IUTs) provide a mechanism for defining OO views capable of resolving semantic heterogeneity among meta-data and data from multiple data sources. Informally, while the DTs represent restrictions (selections) and intersections of extents of other types, the IUTs represent reconciled unions of data in one or more AMOS*II* servers or data sources.

The description of the IUTs in this section is from a perspective of a database administrator who models and defines a mediating view used later by the users. From the users' perspective, there is no difference between querying IUTs and ordinary types. The view definition process will be illustrated by an example of a computer science department (CSD) formed from the faculty members of two universities named *A* and *B*. The CSD administration needs to set up a database of the faculty members of the new department in terms of the databases of the two universities. The faculty members of CSD can be employed by either one of the universities. There are also faculty members employed by the both universities. The full-time members of a department are assigned an office in the department.

One possible system architecture for the data integration problem described above is presented in Figure 5.1. In this figure, the mediators and translators are represented by rectangles; the ovals in the rectangles represent types, while the solid lines represent inheritance relationships between the types. The two translators T_A and T_B provide a representation of the university databases in the CDM of AMOS*II*. In T_A, there is a type *Faculty*

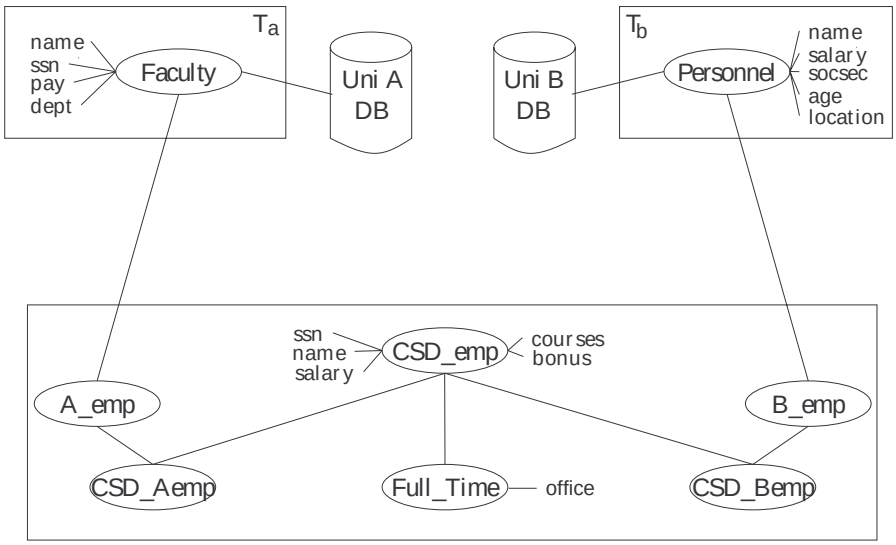


Figure 5.1: An Object-Oriented View for the Computer Science Department Example

and in T_B a type *Personnel*. A mediator is setup in the CSD to provide the integrated view. Here, the types *CSD_A_emp* and *CSD_B_emp* are defined as subtypes of the types in the translators:

```
create derived type CSD_A_emp
  subtype of Faculty@Ta
    where dept(A_emp) = ‘‘CSD’’;

create derived type CSD_B_emp
  subtype of Personnel@Tb
    where location(B_emp) = ‘‘G house’’;
```

The system imports the external types, looks up the functions defined over

them in the originating mediators, and defines local proxy types and functions with the same signature, but no implementation. In this example, the extents of the DTs are specified as subsets of the extents of their supertypes by using simple selections, but in general the subtyping condition can also be joins.

The IUT *CSD_emp* represents all the employees of the CSD. It is defined over the *constituent types* *CSD_A_emp* and *CSD_B_emp*. *CSD_emp* contains one instance for each employee, regardless of whether it appears in one of the constituent types or in both. There are two kinds of functions defined over *CSD_emp*. The functions on the left of the type oval in Figure 5.1 are derived from the functions defined in the constituent types. These *reconciled* functions have more than one overloaded implementation, one for each possible combination of constituent types instances, matching an IUT instance. The functions on the right are locally stored functions.

The data definition facilities of AMOSQL include constructs for defining IUTs as described above. The type *CSD_emp* is defined as follows:

```
CREATE INTEGRATION TYPE csd_emp
  KEYS ssn INTEGER;
  SUPERTYPE OF
    csd_A_emp ae: ssn = ssn(ae);
    csd_B_emp be: ssn = id_to_ssn(id(be));
  FUNCTIONS
    CASE ae
      name = name(ae);
      salary = pay(ae);
    CASE be
      name = name(be);
      salary = salary(be);
    CASE ae, be
      salary = pay(ae) + salary(be);
  PROPERTIES
      courses BAG OF STRING;
      bonus integer;
END;
```

The IUT *csd_emp* definition reveals some details not apparent from the graphical representation of the integration scenario. The first clause defines a set of *keys* and their types. In the example, the key is single valued of

type *integer*. For each of the constituent subtypes, a key expression is given to calculate the value of the key from the instances of this subtype. The instances of different constituent types having the same key values will map into a single IUT instance. The key expressions can contain both local and remote functions.

The `FUNCTIONS` clause defines the reconciled functions of *CSD_emp*, derived from the values of the functions over the constituent types. For different subsets of the constituent types, a reconciled function of an IUT can have different implementations specified in the `CASE` clauses. For example, the definition of *CSD_emp* specifies that the *salary* function is calculated as the salary of the faculty member at the university to which it belongs. In the case when she is employed by both universities, the salary is the sum of the two salaries. When the same function is defined for more than one case, the most specific case applies. If no single most specific case exists (e.g. *name*), the system assumes "any" semantics and chooses one based on a heuristic to improve the performance of the queries over these functions.

Finally, the `PROPERTIES` clause defines the two stored functions over the IUT *CSD_emp*. At any time after the definition of an IUT, the user can add stored or derived functions. The derived functions can be based on any functions already defined in the mediator, regardless whether they are implemented locally or in some other AMOS*II* server.

The IUTs can be subtyped by DTs as any other types. In the example in Figure 5.1, the type *Full_Time* representing the full time employees is defined as a subtype of the type *CSD_emp*. The locally stored function *office* stores the information about the offices of the full time CSD employees.

5.2 Modeling and querying the integration union types

Every instance of an IUT corresponds to either an instance in one of the two constituent types, or to one instance in both of them. Therefore, the extent of an IUT can be divided into three subsets (Figure 5.2a). Two sets contain the IUT instances corresponding to an instance in a single constituent type. The third set contains the IUT instances corresponding to instances in both constituent types. Since the extent subsets can be defined by declarative queries, we can define each of them as a DT, named an *auxiliary type* (AT). The three ATs generated for each IUT form an inheritance hierarchy as

shown in Figure 5.2b.

A function f defined over an IUT can have a different implementation for each of these three subsets (i.e. for each `CASE` clause). It can be thus defined for the whole extent of the IUT by being *overloaded* on the ATs. A call to f for an IUT will then result in a late bound function call, to be discussed below.

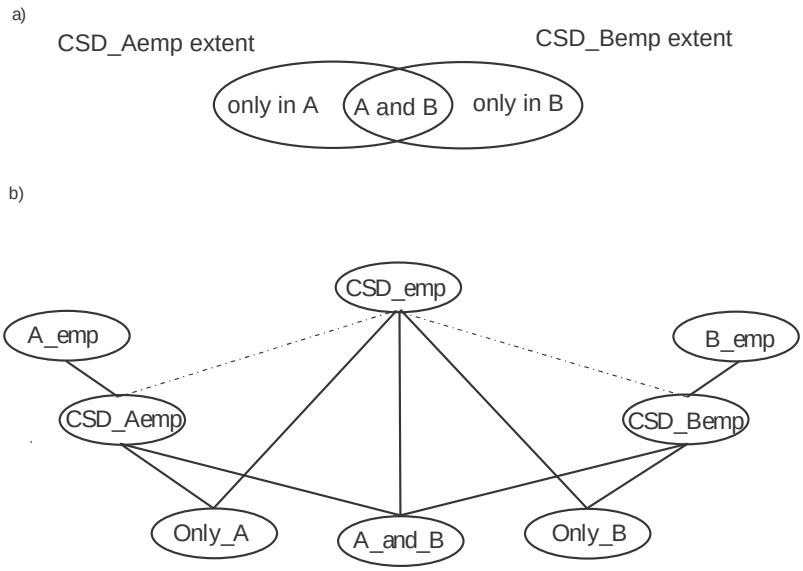


Figure 5.2: IUT implementation by ATs

The ATs are generated by the system and are not visible to the user. Each AT corresponds to a `CASE` clause in the IUT definition. By using the specifications from the `KEYS` clause of the IUT definition, two functions are generated for each constituent type. The overloaded function $key_{CT \rightarrow key\ types}$ calculates the key of an instance of a constituent type CT. The function *Allkeys*`CT`*()* returns all the keys for the type CT. With these functions defined, the AT definitions for the example are:

```
create derived type Only_A
  subtype of CSD_Aemp ae
    where key(ae) not in AllkeysCSD_Bemp();

create derived type Only_B
  subtype of CSD_Bemp be
    where key(be) not in AllKeysCSD_Aemp();

create derived type A_and_B
  subtype of CSD_Aemp ae, CSD_Bemp be
    where key(ae) = key(be);
```

The first two subtypes represent keys based on anti-semi-joins of the integrated types. The third is a join of the integrated types.

Next, the system creates the IUT and makes the ATs its subtypes. The overloaded function resolvents are then defined over the IUT and each of the ATs. The AT resolvents are generated from the *FUNCTIONS* clause in the IUT definition. The resolvent for the IUT itself is defined as *false* since all the instances of the IUT belong to one of the ATs, giving the optimizer a hint to reduce the execution plans.

The extents of the ATs represent mutually exclusive sets of real world entities. The union of these extents forms the extent of the IUT which therefore contains one instance for each entity. From the user's point of view, the only difference between the IUTs and the ordinary types is that no objects can be explicitly created in the IUTs. The extent of the IUTs are completely derived from the extents of the ATs.

5.2.1 Late binding over derived types

To process queries over the system-generated OO views having overloaded functions, we developed a novel late binding mechanism for efficient handling of declarative view definitions in a multiple AMOS*II* servers environment. A late bound function call $f(a)$ is first translated into a calculus *late binding operator* (LBO) whose first argument is a tuple of the possible resolvents of f sorted with the least specific type first, and the second argument is a. For functions used when an IUT is modeled by ATs, the late binding calculus expression is:

$$LBO(< f_{iut}, f_{at1}, ..., f_{atn} >, a)$$

where the ATs $at1 \ldots atn$ are subtypes of iut. Based on the types of the argument a, LBO chooses the most-specific resolvent, executes it over the argument, and returns the result(s).

In our previous work, we have developed a corresponding algebraic late binding operator for the ordinary types, the Dynamic Type Resolver (DTR) [26]. DTR, as most late binding mechanisms described in the literature (e.g. [24]), processes one tuple at a time and selects the query plan of a resolvent based on the type of a. This mode of processing is not suitable for the IUT queries for the following reasons. First, because the resolvents are functions defined over data in multiple sources, processing a tuple at a time results in calling remote functions in an RPC manner. Second, it requires the instances to have assigned OIDs, leading to OID generation for *all* the instances processed in a query, and not only for the ones requested by the user. Furthermore, such a late binding mechanism assumes that the type information of the argument object is explicitly stored with its OID. By contrast, the types in the IUT are defined *implicitly* by queries, and IUT instances can obtain and drop a type dynamically and outside the control of the mediator, based on the state of the data in the sources. Therefore, the use of late binding as above leads into partitioning the query into three separate subqueries: the resolvent function bodies (i.e. the expressions in the `CASE` clauses), the AT subtyping conditions, and the predicate in the query. This separation will prohibit query rewrite techniques from eliminating common subexpressions and other query reduction methods as described in [42] and [23].

In order to overcome these limitations, the LBO is translated into an equivalent disjunctive object calculus predicate, which is then combined and optimized with the rest of the query. AMOS*II* supports multimethods and overloading on all function arguments and the translation algorithm can handle this too. Since the focus of this chapter is the use of these concepts for processing of queries over the IUTs, here we only present a simplified version of the algorithm that handles overloading on a single argument.

In the translated disjunctive calculus expression every branch (disjunct) is a conjunction of a typecheck for an AT and a call to the overloaded function f corresponding to the AT. The translation algorithm is:

```
generate_lb_calculus( resolvents ) -> disjunctive predicate
  result = {res |}; /*empty disjunction predicate */
  while resolvents != ∅   do
```

```
    head = first(resolvents);
    /* the argument type for the head function */
    t_h = arg_type(head);
    if ∄f ∈ resolvents | subtype_of(argtype(f), t_h) then
      result = append(result, ∨{arg = t_h() ∧ res = f_{t_h}(arg)});
    else
      wset = {t_p | subtype_of(t_p, t_h) ∧
                  ∄f ∈ resolvents | subtype_of(t_p, argtype(f))}
      for each t_p in wset
        result = append(result,
                        ∨{arg = Shallow_t_p(), res = f_{t_h}(arg)}));
    end if
    resolvents = resolvents - head;
  end while
  return result;
end;
```

First, *append*, and $-$ perform the usual set operations, and *arg_type* returns the argument type of a function. The algorithm traverses the sorted list of resolvents. If the type hierarchy rooted in the argument type of a resolvent does not intersect with the hierarchies of the argument types of some resolvents in the rest of the list, then a conjunction of an ordinary (deep) typecheck and the resolvent call is added as a new disjunct to the result. Otherwise the new disjunct will instead contain a shallow typecheck. Notice that for IUTs there will be no shallow typechecks, because there are never any subtypes of the system-generated ATs. Since the type checks are mutually exclusive, only one resolvent will be evaluated.

To illustrate the translation process we examine the translation of the LBO for the function *salary* over the IUT *CSD_emp*:

$$LBO(< salary_{csd\_emp \to int}, salary_{Only\_A \to int},$$
$$salary_{Only\_B \to int}, salary_{A\_and\_B \to int} >, arg)$$

is translated into:

$$\{\ s \mid$$
$$(arg = only\_A_{nil \to only\_a}() \ \wedge\ s = salary_{only\_A}(arg)) \ \vee$$
$$(arg = only\_B_{nil \to only\_b}() \ \wedge\ s = salary_{only\_B}(arg)) \ \vee$$
$$(arg = A\_and\_B_{nil \to a\_and\_b}() \ \wedge\ s = salary_{a\_and\_b}(arg))\}$$

The expression is a disjunction of only three disjuncts. No disjunct is generated for the first resolvent $salary_{csd\_emp \rightarrow int}$ since it is defined as *false*.

After the query normalization, the extent functions of the ATs are expanded by substituting them with their bodies containing the expressions from the `CASE` clauses of the IUT definition. These expressions in turn reference the extent functions of the constituent types, which are DTs and the expansion continues until no DT extent functions are present. This process makes visible to the query decomposer i) the query selections defined by the user, ii) the conditions in the IUT, and iii) the DT definitions. The query decomposer combines the predicates, divides them into groups of predicates executable at a single mediator, translator or data source, and then schedules their execution. As opposed to dealing with parametric queries over multiple databases, as would have been the case with a tuple-at-the-time implementation of the late binding, the strategy ships and processes data among the mediators, translators, and data sources in bulks containing many tuples. The size of a bulk is determined by the query optimizer to maximize the network and resource utilization. The results in the next section demonstrate how the bulk-processing allows for query processing strategies with substantially better performance than the instance-at-the-time strategies. Furthermore, this strategy allows the optimizer to detect and remove unnecessary OID generations for the instances not in the query result.

5.2.2 Normalization of queries over the integration union types

If there are disjunctive predicates, we need to normalize the query to disjunctive normal form in order to separate the subqueries for the individual data sources. One drawback of the query normalization is that it duplicates predicates in several different disjuncts of the normalized disjunctive predicate. To avoid some of the unnecessary duplication, we use a query normalization which is aware of the multidatabase environment. The normalization algorithm is based on the principle that as many as possible of the normalization decisions should be delegated to the sites where the predicates are executed. Therefore the query decomposer analyzes the elements of a disjunctive predicate and groups together the disjuncts executed in the same mediator, translator, or data source capable of processing disjunctions.

Another source of disjunctions in queries over IUTs are the late bound functions from above, which are translated to disjunctions. A full disjunctive

normalization would then produce a cross product of the disjuncts in all the late bound IUT functions. For example the query:

```
select salary(e), ssn(e) from csd_emp e;
```

produces the calculus expression:

$$
\begin{aligned}
&\{\ sal, ssn\ \mid \\
&\ (arg = only\_A_{nil \to only\_a}()\ \wedge\ sal = salary_{only\_A}(arg))\ \ \vee \\
&\ (arg = only\_B_{nil \to only\_b}()\ \wedge\ sal = salary_{only\_B}(arg))\ \ \vee \\
&\ (arg = A\_and\_B_{nil \to a\_and\_b}()\ \wedge\ sal = salary_{a\_and\_b}(arg))\ \wedge \\
&\ (arg = only\_A_{nil \to only\_a}()\ \wedge\ ssn = ssn_{only\_A}(arg))\ \ \vee \\
&\ (arg = only\_B_{nil \to only\_b}()\ \wedge\ ssn = ssn_{only\_B}(arg))\ \ \vee \\
&\ (arg = A\_and\_B_{nil \to a\_and\_b}()\ \wedge\ ssn = salary_{a\_and\_b}(arg))\}
\end{aligned}
$$

The expression is then normalized into 9 disjuncts, one for each combination of the disjuncts in the two disjunctive predicates above. This expression shows the first two disjuncts:

$$
\begin{aligned}
&\{\ sal, ssn\ \mid \\
&\ (arg = only\_A_{nil \to only\_a}()\ \wedge\ sal = salary_{only\_A}(arg)\ \wedge \\
&\ \ arg = only\_A_{nil \to only\_a}()\ \wedge\ ssn = ssn_{only\_A}(arg))\ \ \vee \\
&\ (arg = only\_B_{nil \to only\_b}()\ \wedge\ sal = salary_{only\_B}(arg)\ \wedge \\
&\ \ arg = only\_A_{nil \to only\_a}()\ \wedge\ ssn = ssn_{only\_a}(arg))\ \ \vee\ \ \ldots\}
\end{aligned}
$$

We can see that each disjunct contains two typecheck predicates for the variable *arg*. This will also be the case in the remaining six disjuncts not shown above. Based on the presence of more than one typecheck over the same variable in a conjunctive predicate and on the properties of the type hierarchy, the disjuncts generated by the query normalization can be rewritten into a simpler form or eliminated.

Since an object can have only one most specific type, two typecheck predicates for a single variable of two unrelated types are always rewritten to $false$, and the disjunct is removed. When the types are related, depending on whether the typechecks are deep or shallow, the result of the rewrite is either $false$ or the more specific typecheck predicate.

These rewrite rules eliminate in the example above all six disjuncts in which the typecheck is not performed over the same type (they remove the second of the two disjuncts shown above). In the remaining three it leaves just a single typecheck predicate transforming the query into the following

predicate which will be shown to be significantly faster than the original query:

$$
\begin{aligned}
&\{\ sal, ssn\ \mid \\
&\ (arg = only\_a_{nil \rightarrow only\_a}() \wedge \\
&\ \ sal = salary_{only\_a}(arg) \wedge \\
&\ \ ssn = ssn_{only\_a}(arg)) \vee \\
&\ (arg = only\_b_{nil \rightarrow only\_b}() \wedge \\
&\ \ sal = salary_{only\_b}(arg) \wedge \\
&\ \ ssn = ssn_{only\_b}(arg)) \vee \\
&\ (arg = a\_and\_b_{nil \rightarrow a\_and\_b}() \wedge \\
&\ \ sal = salary_{a\_and\_b}(arg) \wedge \\
&\ \ ssn = ssn_{a\_and\_b}(arg))\}
\end{aligned}
$$

5.2.3 Managing OIDs for the IUTs

The IUT instances are assigned OIDs when used in locally stored functions. For example, a query giving a bonus of $1000 to all employees in the department with salary lower than $1000 can be specified as:

```
set bonus(csde) = 1000 from CSD_emp csde
                       where salary(csde) < 1000;
```

In order to manipulate the IUT OIDs we have generalized the framework developed for handling OIDs of DT instances presented in the previous chapter to the IUTs. As noted previously, the DT functionality is modeled with three functions: the OID generation function, the extent function, and the validation function. Next we describe how the system generates each of these functions for the IUTs.

Since an IUT is a supertype of the corresponding ATs, every AT instance is also an instance of the IUT. Each distinct real-world entity is always represented by an instance in exactly one of the ATs. Therefore, the extent of an IUT is a non-overlapping union of the extents of the ATs and the extent function of an IUT is a disjunction of the extent functions of its ATs.

The OID generation function assigns an OID to a DT instance. In the case of DTs, the OID generation function is called by the extent function. Since the extent function of an IUT only references the extent functions of its ATs, there is no need for OID generation functions for IUTs. The IUT

instances are thus assigned OIDs by the OID generation functions of the ATs.

If the ATs were treated as ordinary DTs, the assignment of OIDs to the AT instances would be made independently of the other ATs of an IUT. On the other hand, due to the nature of the conditions used in the ATs definition, instances 'drift' from one AT to another. For example, let's assume that John Doe is an employee of University A, and also a member of the CSD in the example above. When his bonus is assigned, the system will generate an OID for the instance representing John Doe in the AT *Only_A* and use this OID in the stored function *bonus* to relate John with his bonus. If John now gets an appointment at University B, he still belongs to the *CSD_emp* IUT, but an instance representing him appears in the type *A_and_B*, while the instance in the type *Only_A* is removed. If the newly created instance in *A_and_B* has a different OID from the old instance in *Only_A*, then John cannot be matched with his bonus stored in the database using the old OID.

The example shows that the OID assignment for instances of the ATs must be coordinated, so the instances representing the same real-world entity can move from one AT to another, while preserving their identity. An instance is related to a real world entity through its key, so to solve the problem, the OID assignments of the ATs are controlled by a function storing the generated OIDs along with the keys. When a new AT OID is to be generated, the OID generation function first checks if there is a stored OID with a matching key. If so, it adjusts the type of the stored OID and returns it as result. Otherwise, it generates a new OID. We notice here that, because the selections are pushed to the data sources and due to the OID generation removal mechanism described in chapter 4, only a subset of the whole IUT extent is assigned OIDs in queries containing selections. Very often, queries require function values and not the OIDs of the queried types. In these cases no OIDs will be generated at all.

In chapter 3 an example was presented on how the typecheck predicate of a variable can be removed from a query when the variable is used in a predicate with a locally stored function of that type. This mechanism, described in greater detail in [52], is extended to apply over the IUTs. An advantage of removing the typecheck is that the costly generation of the IUT extent is not needed, but instead only the already generated OIDs stored in the local function are used. However, when dealing with stored DT or IUT instances, we need to make sure that they are still valid, i.e. that the data sources still contain the corresponding instances.

A straightforward solution to the problem of validating an IUT instance is to test which of the three IUT ATs it belongs to. It is, however, sufficient to validate an IUT instance by testing the existence of a corresponding instance having the same key in one of the two integrated sources; the intersection AT need not be tested. This condition can be expressed by a two-branch disjunctive predicate instead of a three-branch one in the straightforward solution. The gain is due to the fact that we are not interested in exactly which AT an IUT instance belongs to, but if it belongs to *any* of the ATs. As an example we present the calculus representation of the validation function body for the *CSD_emp* type from the example above:

$$
\begin{aligned}
&validate_{csd\_emp}(e) \leftarrow \\
&(ssn = skey_{csd\_emp \rightarrow integer}(e) \wedge \\
&\ ssn = ssn_{csd\_a\_emp \rightarrow integer}(csda)) \ \vee \\
&\\
&(ssn = skey_{csd\_emp \rightarrow integer}(e) \wedge \\
&\ id = id_{csd\_b\_emp \rightarrow string}(csdb) \wedge \\
&\ ssn = id\_to\_ssn_{string \rightarrow integer}(id))
\end{aligned}
$$

The variables *csdb* and *csda* are local variables.

The validation method described above suffices when a query contains only locally stored functions over an IUT, while not containing late bound functions over the same IUT. When a query contains both locally stored and late bound functions, the system needs to determine which AT an IUT instance belongs to, in order to execute the right resolvent. Since an instance can drift between the ATs, the system must determine the AT membership for the IUT instances at query time. In order to do this, a disjunctive predicate similar to the one described earlier in this section is used. The only difference is that here the typecheck predicates are replaced with the corresponding validation predicates.

5.3 Performance measurements

The AMOS*II* system with the mediation features described in this thesis is implemented on Windows NT. We will present an overview of some experimental results obtained from running the system over 10Mb Ethernet and ISDN networks. The results demonstrate how the techniques presented above drastically reduce the response times.

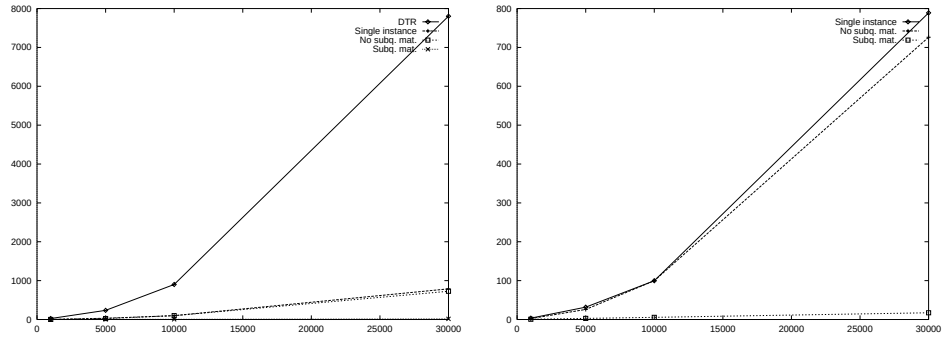
The experiments are performed for a scenario similar to the running example above. We used two Compaq Professional Workstation 5000 with 200MHz Pentium processors and 64 MB memory, connected through a 10Mb Ethernet network. We also performed the same tests using a 64kb ISDN connection over the public telephone network in Sweden.

One of the workstations hosted an ODBC data source and an associated AMOS*II* system as a translator. For the experiments we used Microsoft Access as a relational data source because of its availability, but the results apply to any other ODBC data source. On the second workstation, another AMOS*II* server represented another data source. To be able to quantify the difference in the times between the processing in AMOS*II* and in the ODBC data source, the data was here stored directly in the AMOS*II*'s main-memory database. The second workstation also hosted the mediator system where the queries were issued. The three AMOS*II* servers just described will be referred to in the rest of this section as T_a (the ODBC translator), T_b (the AMOS*II* storing data locally) and the *mediator* for the AMOS*II* server where the queries are issued.

In the experiments, we scaled simultaneously the tables *Faculty* in the ODBC data source and the extent of the type *Personnel* stored in T_b from 1000 to 30000 tuples. From these tuples, 10% are selected to be members of each of the types *A_emp* and *B_emp* (i.e. members of the CSD), which are the constituent types for the integration type *CSD_emp*. Between these two types, we assume that half of the instances are overlapping (represent the same persons), meaning that the size of the extent of the type *CSD_emp* is 15% of the cardinality of the table. For example, when the size of both the *Faculty* table in the ODBC source and the extent of the type *Personnel* in T_b is 30000, there are 3000 instances of each selected as working in the CSD department by the conditions in the definition of the derived types *A_emp* and *B_emp*. From each of these two sets of 3000 instances, 1500 appear only in one of these types and 1500 appear in the both constituent types. The extent of integrated type *CSD_emp* therefore has 4500 instances.

The experiments are based on queries over the IUT *CSD_emp*. The queries are simple in order to analyze certain features of the system. Also, we have chosen queries that are the building blocks of most user-specified queries over the IUTs. More specifically the test cases can be divided into i) queries over reconciled IUT functions, and ii) queries calling locally stored functions over the IUT. In the former group we first investigate queries with no selection, exact match, and range selections. Then we present results when more

than one function is used in the same query, to investigate the performance impact of the type-aware rewrites. Queries with locally stored functions are investigated in one example. We conclude the tests by comparing the times for some queries over the 10Mb network with the times obtained when the same queries were executed over an ISDN network. Notice that the y-axis in all the graphs represents response time in seconds and the x-axis represents the number of tuples in the test databases. All the measurements are performed with preoptimized queries.



(a) DTR strategy

(b) Other strategies

Figure 5.3: Query: select salary(e) from csd_emp e;

Figure 5.3 shows the execution time of a query retrieving the salaries of the CSD employees. We examine 4 different strategies. The graph on the left shows that the "DTR" strategy using pure late binding on an instance level is by orders of magnitude worse that the remaining three strategies. This strategy, first generates OIDs for all instances in the extent of the type CSD_emp. Then, DTR is executed over each of the OIDs, choosing the resolvent. Finally, the chosen resolvent is executed. The resolvent body also contains predicates to confirm the right AT of the argument, which causes the typecheck to be executed once again before the function value is calculated.

The left part of Table 1 shows the percentage of the time spent in the three cooperating AMOS*II* servers, and the network time for each of the

	Time distribution			
	Mediator	T_a	T_b	Net.
DTR	23%	69%	1%	7%
Single instance	5%	80%	3%	12%
No subq. mat.	3%	91%	3%	3%
Subq. mat.	27%	22%	32%	19%

Table 5.1: Query execution time distribution for the 4 evaluation strategies

examined strategies. For the DTR strategy, the biggest portion of the query execution time is spent in T_a for accessing the relational data source. Table 5.2 presents the number of ODBC calls issued by the data source T_a for the different strategies. The DTR strategy issues by far the most of such calls. The number of calls is a linear function of the data sizes in the sources, but as the data volume grows, each of these calls demands more time, explaining the hyper-linear growth in the query execution time. We can also note that the DTR strategy spends 23% of the time in the mediator. This is due to OID generation, function resolution, and execution of the protocol for shipping instances among different AMOS*II* servers. The OID generation for IUT instances requires that OIDs are generated for the constituent types, which in turn triggers proxy object generation for the instances imported from the translators. Since the DTR operator is executed over each instance individually, a large amount of computation is involved.

The lower part of the graph in Figure 5.3a is enlarged in Figure 5.3b. Here, we can see the remaining 3 query processing strategies. The uppermost curve represents a strategy in which the late bound function call is substituted by a disjunctive predicate, but the data shipment is still one instance at the time. This type of nested loop join over a network is named bind-join in [36]. Query rewrites eliminate OID generation, duplicate condition evaluations, and run-time function resolution. Also, the number of ODBC calls in T_a is reduced by two thirds. All of this reduces the query execution time by nearly 10 times. Nevertheless, the ODBC calls are still the main factor in the query execution cost. We can also note that the relative network cost has risen to 12%.

The first step into designing a better strategy is to pass the instances in bulks instead of an instance-at-a-time protocol. While this strategy, due to the fast networks used, does not radically improve the result (the next curve in the graph in Figure 5.3b), it does lower the relative network cost to 3%

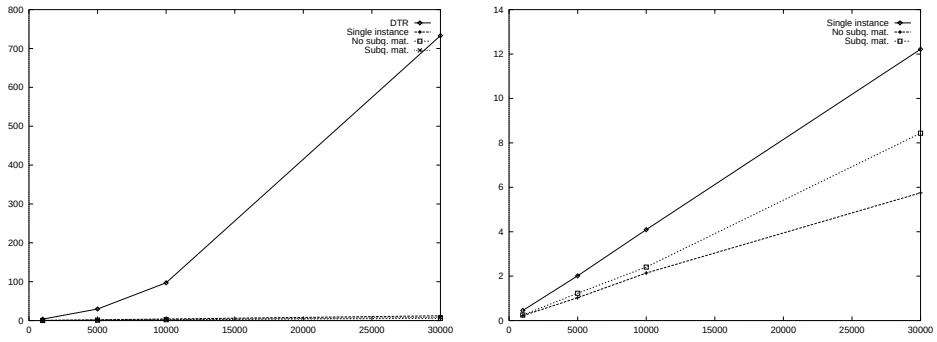
and makes the final query strategy possible.

	ODBC requests / DB size			
	1000 t.	5000 t.	10000 t.	30000 t.
DTR	251	1251	3001	9017
Single instance	102	502	1002	3002
No subq. mat.	102	502	1002	3002
Subq. mat.	3	3	3	6

Table 5.2: Number of data source accesses for the 4 evaluation strategies

The final strategy, which again reduces the response time by a couple of orders of magnitude, is based on the observations that most of the ODBC queries are issued to compute the extents of the ATs which involve anti-semi-joins translated into nested subqueries inside a *not exists* operator. In order to avoid the cost of repeated data access using parametric queries, we execute a single non-parametric query and materialize an index over all the parameter values in T_a. In this example the index contains the *ssn* for the 10% employees of University A who are also in CSD. In this way, we reduce the ODBC requests to one per bulk sent from the mediator to the translator. Being a main-memory based database, AMOS*II* facilitates a very fast index build-up for data sizes which can fit into the memory of the translator. For this type of query where the materialized index is used repeatedly, this strategy is clearly advantageous. We can also see that the distribution of the query execution time in the last strategy is balanced evenly among the participating AMOS*II* servers and the network. Note that there is one access to the data source per disjunction branch of the query. Therefore, the skew in the data distribution will not affect the query execution times.

The cost of executing a non-parametric query and building an index on-the-fly has to be compared with the cost of completing the query without the index. In the next experiment, we executed a query containing an exact match selection using the same 4 strategies. The DTR strategy is again by far the worst, as shown in Figure 5.4a. On the other hand, the differences among the other strategies is not as large as in the previous experiment (Figure 5.4b). Also, here the strategy without index materialization for the nested subquery performs the best. This is due to the fact that the non-parameterized query used to compute



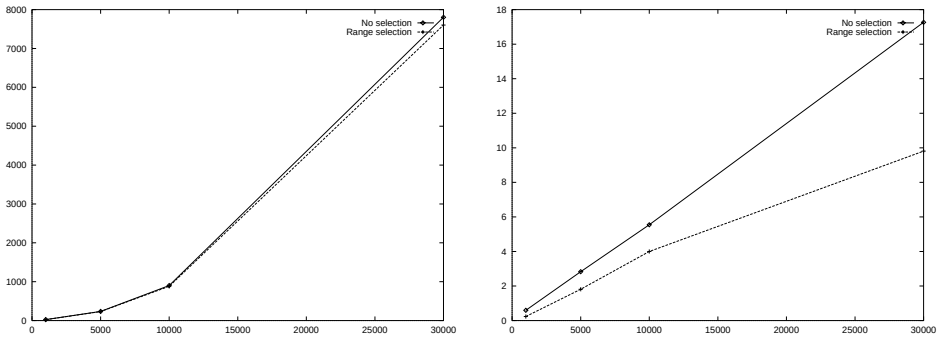
(a) DTR strategy (b) Other strategies

Figure 5.4: Query: select salary(e) from csd_emp e where ssn(e) = 1000;

the index has a larger cost than the parameterized query retrieving only the data matching a particular input tuple. In general the index materialization is favorable when: $size(input) * cost(parameterized\ query) > cost(non\ parameterized\ query) + cost(index\ generation)$.

In the next experiment we examine queries with non-equality selections, e.g. range selections. While the DTR strategy is able to apply the selections encapsulated in the DT condition, it is not efficient when the query contains non-equality conditions, since such conditions are then not pushed into the resolvents. In Figure 5.5a the execution times of a query containing a range selection is compared with the execution times of a query without any selection. It can be seen that the cost is about equal. In Figure 5.5b, on the other hand, there is clear difference between the execution times of the same queries using disjunctive predicates to model the late binding. This is due to the fact that the selection is pushed all the way down to the data sources.

Next, we measure the execution time for queries containing locally stored functions over an IUT. In this experiment, we created a locally stored function *office* over the type *CSD_emp* storing only 15 rows, and then executed a query to retrieve the offices stored in this function. Figure 5.6a compares the execution times of a naive strategy where the system generates the OIDs for the type extent and then applies the locally stored function with the strategy where the IUT instances of interest are retrieved from the locally stored function and then validated as described previously. Since the cardinality of



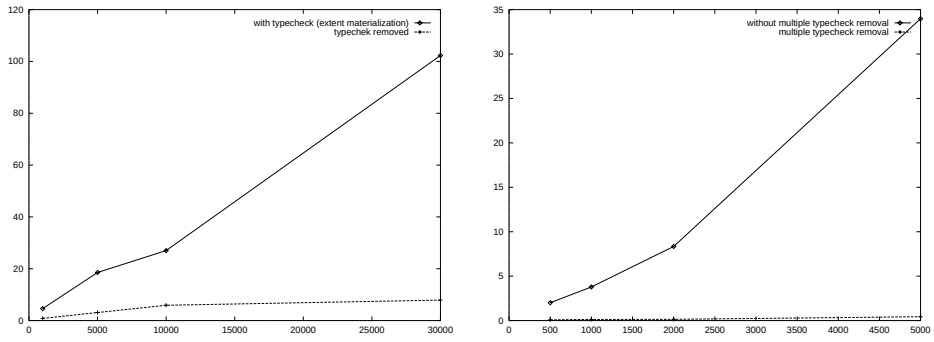
(a) DTR strategies

(b) disjunctive pred. with subquery materialization strategies

Figure 5.5: Selecting salary for the CSD employees with and without range selection (salary(e) > 2000)

a locally stored function is always smaller than the cardinality of the whole type extent, and the validation of an already generated OID is cheaper than a new OID generation, the validation strategy always outperforms the naive strategy.

The graph in Figure 5.6b demonstrates the speedup obtained by type-check removal using type-aware rewrites described in the previous section. The query is normalized to a disjunction with 9 branches, 6 of which are removed by the optimizer. The execution times on the other hand show greater than linear speedup and scalability as could be expected from the analysis of the number of the disjunctive branches. This is due to the fact that the 3 remaining branches after the query transformation are single type queries with a selection condition. The rest of the 6 queries are effectively join queries over different ATs. In these cases, the AT extent functions and the extent functions of the constituent types are expanded for both the ATs appearing in the typecheck predicates. The optimizer cannot infer on the basis of these predicates that the whole disjunct will not produce any results. The resulting query execution strategy cannot therefore take advantage of the selections, and ships data proportional to the size of the extents of the constituent types. This leads to execution times with linear growth with the size of the extents, as opposed to the much slower growth of the execution



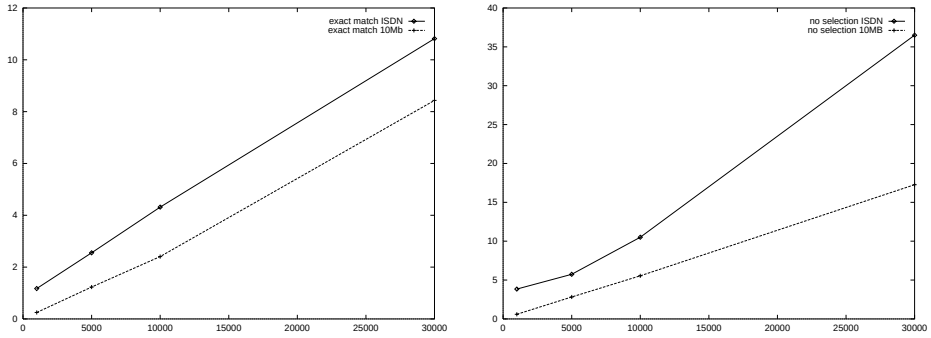
(a) select office(e) from csd_emp e;

(b) select salary(e), name(e) from csd_emp e where name(e)="John";

Figure 5.6: a) Queries with locally materialized functions over IUTs. b) Queries calling several derived functions over IUTs.

time when the rewrite rule for removal of the typechecks is applied.

Finally, we briefly compare the execution times obtained over a 10Mb network with the results of the experiments using an ISDN connection over a public telephone network. Keeping all the parameters of the testing the same, the difference in the times can be attributed to the properties of the networks. The graph in Figure 5.7a shows that when the number of the manipulated tuples is low, the results are proportional. However, when the amount of shipped tuples increases, as with the query without selection used in Figure 5.7b, the execution times over ISDN rise faster than over the 10Mb network. Closer examination revealed that ISDN execution times follow the number of data bulks sent over the network. We can conclude that the unproportional increase is due to the fact that the message setup time compared to the transmission time per unit is higher in ISDN networks than it is in the 10Mb Ethernet. The optimizer should probe the network and determine the bulking factor according to these parameters. This will be one of the topics of future investigation.



(a) Exact match selection

(b) No selection

Figure 5.7: Comparison of execution times over a 10Mb network with an ISDN network.

Chapter 6

Query Decomposition and Execution

This chapter describes the second functional unit of the data integration facilities in AMOS*II*: the multidatabase query processor. It consists of two units:

- Query decomposer
- Query execution run-time support

The goal of the query decomposition is, given a query over multiple data sources, to search the space of possible execution schedules and choose a "reasonably" cheap one. The run-time support provides protocols for efficient execution of the schedules produced by the query decomposer. The query decomposer and the query run-time support are closely related to each other. The query decomposition estimates query schedule costs based on the properties of the applied execution algorithms. The run-time support takes as input an algebra plan generated by the query decomposer.

As noted earlier, AMOS*II* is a distributed mediator system. This implies a framework that allows cooperation of a number of distinct AMOS*II* servers on a query processor level. While distribution is present in any mediation framework due to the distribution of the data sources, the distributed mediator framework in AMOS*II* introduces a higher level of interaction among the AMOS*II* servers. In other words, an AMOS*II* server does not treat another AMOS*II* server as just another data source. More specifically, if we compare

the interaction between an AMOS*II* system and a wrapper (and through it with a data source), and the interaction between two AMOS*II* servers, there are two major differences:

- AMOS*II* can accept compilation and execution requests for subqueries over data in more than one data source, as well as data stored in the local AMOS*II* database. The wrapper interfaces accept queries that are always over data in a single data source.

- AMOS*II* supports materialization of intermediate results to be used as input to locally executed subqueries, generated by a query decomposition in another AMOS*II* server. A wrapper provides only *execute* functionality for queries to the data source. The query execution interface of AMOS*II*, on the other hand, provides *ship-and-execute* functionality that can first accept and store locally an intermediate result, and then execute a subquery using it as an input.

These two features influence the design of both the query decomposer and the run-time support for query execution. Techniques based on these features are used in the work presented in this chapter to achieve improved query performance. The remaining of the chapter describes first the query decomposition process, and then the query execution and run-time support in AMOS*II*.

6.1 Query decomposition

The query decomposition produces an executable algebra plan from a parsed, and flattened query calculus expression. The parsing and the flattening of the multidatabase queries are not different from the parsing and the flattening of the other queries. The query decomposition process is performed in 5 phases:

1. Predicate grouping
2. Execution site assignment
3. Execution schedule generation
4. Tree rebalancing and distribution
5. Object algebra generation

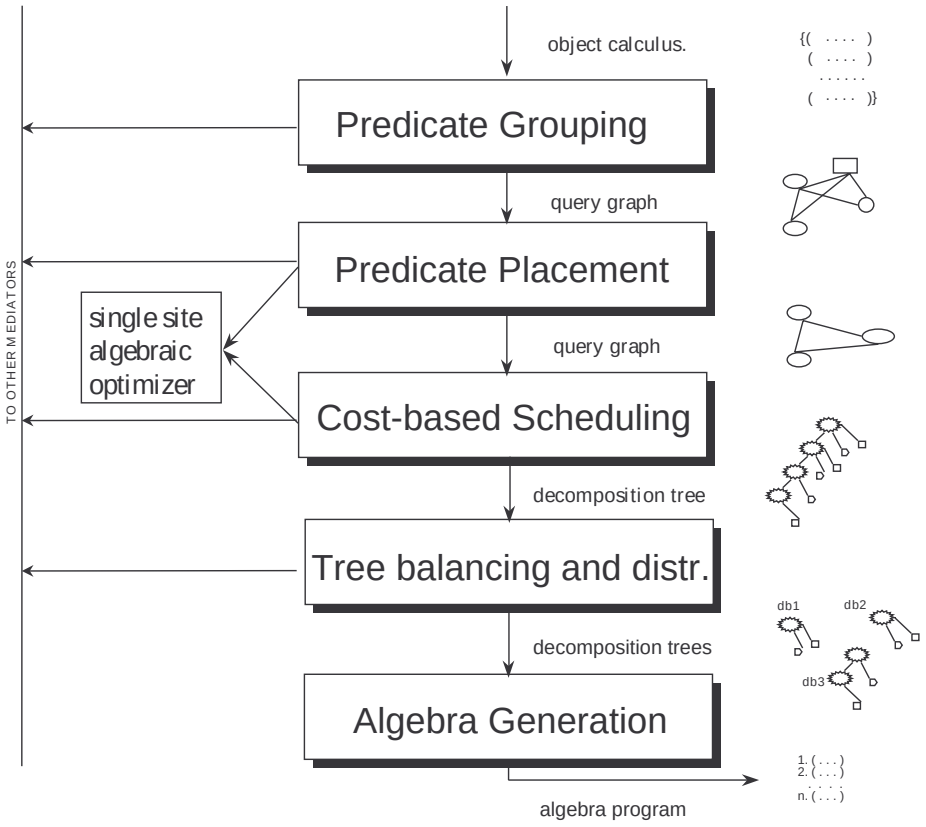


Figure 6.1: Query Decomposition Phases in AMOS*II*

Figure 6.1 illustrates the individual query decomposition phases and their results.

To approximate the hardness of the problem of finding the optimal query execution plan for a calculus expression over multiple data sources, we could represent the possible query execution plans as n-ary operator trees. However, this formalism is not used in AMOS*II*, its purpose is solely to demonstrate the enormous size of the search space of this optimization problem.

Each tree node contains a simple predicate from the query calculus expression, and is assigned for execution at a data source. Some predicates can be executed at more than one data source. A tree is executed by first executing the root node children, then shipping the results to the site (data source) where the root node is assigned, and finally executing the root node predicate. Since the number of possible n-ary trees with p nodes is exponential to p, and the number of different site assignments is exponential to the number of predicates executable at more than one data source d, the total number of trees is $O(a^p)O(s^d)$, where a is a constant and s is the number of sites involved. This estimate shows that an exhaustive search of the whole search space is not feasible. Therefore the decomposition strategy in this work combines cost-based search strategies with heuristic rules to lower the number of the examined alternatives.

The description of the query decomposition in this section assumes conjunctive predicate expressions as input. The query decomposer handles disjunctions in two ways, depending on the origin of the predicates in the disjuncts:

- **single source disjunctions** containing predicates that are executed at a single data source are treated as a single predicate with a cost, a selectivity and a binding pattern induced form the disjuncts.
- **multiple source disjunctions** are handled by normalization of the queries into disjunctive normal form. The decomposer then processes each of the disjuncts in the normalized query separately.

The rest of this section describes the decomposition phases in greater detail. First, in order to provide a basis for this description, the support for data sources with different capabilities in AMOS*II* is presented.

6.1.1 Data source types and functions with multiple implementations

While some of the functions used in the AMOSQL queries are implemented, and can be executed, in exactly one data source, there are also functions that can be executed in more than one data source. According to the this criteria, the functions in AMOS*II* are classified into:

- *single implementation functions* (SIFs)
- *multiple implementations functions* (MIFs)

The user-defined local functions as well as the proxy functions are single implementation functions. For example, the function $name_{Person \rightarrow string}$ is a SIF, defined over the instances of the type $Person$ in $EMPLOYEE\_DB$. The implementation of this function is known only in that mediator and therefore it can be executed only there. The second category contains functions that are executable in more than one data source, as for example, the comparison operators (e.g. $<$, $>$, etc.) that are executable in AMOS*II* servers, relational databases, certain storage managers, etc. The MIFs can also be user-defined. However, since in our framework each user-defined type is defined in only one data source, a MIF may take only literal typed arguments. A framework that would support replicated user-defined types and MIFs taking user-defined type arguments would require that the state (value) of the instances of these types is shipped among the mediators, in order to be used at the data source where the MIF is executed. In the framework presented in this thesis, only OIDs and the needed portions of the instances' state is shipped among the mediators and the data sources. Replicated user-defined types can be simulated by stringifying the state of the instances and handling them in the mediators as character strings. The wrappers translate the stringified instances from and to the representations required in the data sources. Extending the integration framework to handle replicated user-defined types is one of the topics of our current research.

Depending on the set of MIFs implemented at a data source, the data sources are classified into several *data source types* (DSTs). Inversely, the set of MIFs associated with a DST is named *generic capability* of the DST. Besides a generic capability defined by its type, each data source can have *specific capability* defined by the types and functions exported to the AMOS*II* mediators or translators. To simplify the presentation, in the rest of this chapter the term *capability* is used to refer only to a generic capability of a source or a DST.

In order to reuse the capability specifications, the DSTs are organized in a hierarchy where the more specific DSTs inherit the capabilities of the more general ones. This hierarchy is separate from the AMOS*II* type hierarchy and is used only during the query decomposition as described below. Figure 6.2 shows an example of an AMOS*II* DST hierarchy. All DST hierarchies are rooted in a node representing data sources with only the basic capability to execute one simple calculus predicate that invokes a single function/operation in this source and returns the result to the translator. This corresponds to a sequential *scan* capability in some other mediation

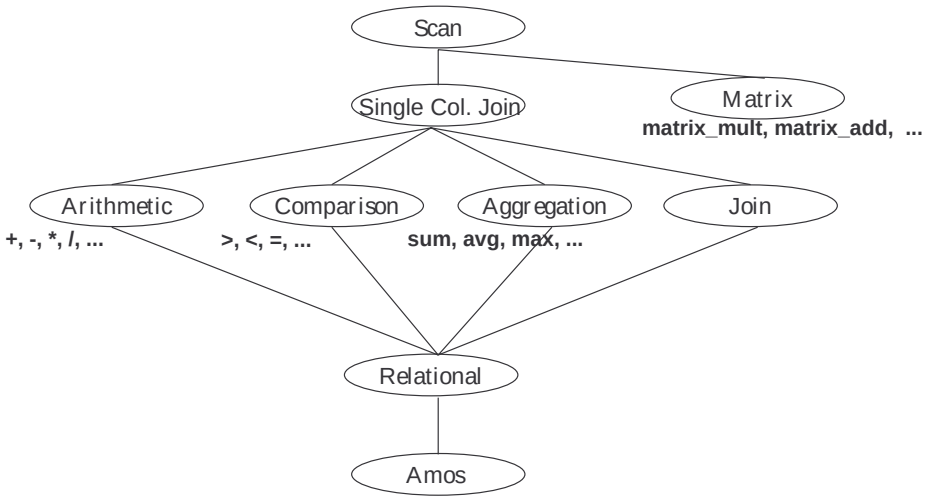


Figure 6.2: Data source capabilities hierarchy

frameworks [36]. Data sources of this type cannot execute MIFs. At the next capability level, DSTs with capabilities to perform arithmetic, comparison and join operations are defined. The arithmetic and comparison DST are defined using the usual set of MIFs, shown in the figure. A MIF in a capability of a certain DST can be defined as a generic function, when all of its resolvents are executable at the sources of the specified DST, or as a specific resolvent when only a particular resolvent can be scheduled for execution at the specified type of sources.

The two join capabilities, the *single collection join* (SC join) and the general *join*, are not specified using MIFs as all the other DST capabilities. In the calculus used in AMOS*II*, the equi-joins are represented implicitly by a variable appearing in more than one query predicate. Accordingly, the join capabilities represent that a data source (and its wrapper) can handle several predicates connected by common variables as a single unit. The predicates executed in such data sources can be grouped together before sending them to the wrapper to achieve more efficient translation to expressions in the local language. For example, relational databases and AMOS*II* servers can perform joins, and therefore it might be favorable to allow join operations

to be pushed to data sources of these types.

Based on the properties of the commonly used data sources, there is a need to distinguish between two types of join capabilities. First, there are sources that are capable of combining and executing conditions over only a single collection of data items in the source (e.g. a table in a storage manager). These types of sources are defined by using a DST that inherits only the SC join capability. An example of this kind of a DST is a storage manager storing several data tables, each represented in the AMOS*II* translator as a proxy type. Each table can be scanned with associated conditions. The conditions to a single table can be added together. However, operations over different tables need to be submitted separately. Therefore, for each table, the MIF operations are grouped together with the proxy type typecheck predicate, and submitted to the wrapper. One such grouped predicate is submitted for each different collection. A system with these types of properties fits the capability description of the *comparison* DST in figure 6.2.

The general *join* capability is inherited by DSTs capable of processing joins over multiple collections (e.g. relational database sources). The decomposer sees each collection as a proxy type, and together with a join capability, it combines the operations over several proxy types into a single subquery sent to the data sources.

New DSTs are defined by inserting them into the DST hierarchy. For example a DST representing a software capable of matrix operations is named *Matrix*, and placed under the DST hierarchy root node in figure 6.2. This implies that it supports the execution of one operation at a time. A source that allows a combination of several matrix operations would have been defined as a child of the *Join* DST.

6.1.2 The predicate grouping phase

The predicate grouping phase attempts to reduce the optimization problem by reducing the number of predicates. In this phase, if possible, the predicates executed at a given data source are grouped into one or more *composite predicates*, treated afterwards as a single predicate. Within each composite predicate, the optimization is performed in the wrapper or the data source where the predicate is forwarded for execution. For each composite predicate, a temporary derived function is defined locally or at some other AMOS*II* server if the predicate consists of proxy functions imported from another

AMOS*II* server [1]. In the query, each predicate group is substituted by a predicate calling the corresponding derived function. The arguments of these functions are the calculus variables appearing in the predicate and in the rest of the query. The types of the function arguments are deduced from the function signatures used in the query predicate. Two major challenges arise in the predicate grouping:

- **Grouping heuristic**: an exhaustive approach to the grouping would not reduce the query optimization problem. A heuristic approach must be used.
- **Grouping of the MIF predicates**: how to group the MIF predicates given different data source capabilities.

The following grouping heuristics are used in AMOS*II*:

- Joins are pushed into the data sources whenever possible.
- Cross-products are avoided.

The grouping process is performed using an undirected graph built from the predicates in the query, called *query graph*, and similar to the query graphs used in centralized database query processors. The initial query graph contains one node for each equality predicate in the flattened query calculus representation. Nodes whose predicates contain common variable(s) are connected by an edge. Each edge is labeled with the variable(s) it represents. The variables labeling the edges connecting a node with the rest of the graph are named *node arguments*.

Nodes representing SIF predicates are named *SIF nodes*. Similarly, the rest of the nodes are named *MIF nodes*. All graph nodes are assigned to a site[2]. The SIF nodes are assigned to the site of their predicates. MIF nodes are assigned to a site in the later decomposition phases. The graph nodes are also assigned a DST. The SIF nodes are assigned the DST of the site where they are to be executed. The MIF nodes are assigned a DST on the basis of the function in the predicate.

[1] A derived function contains, beside a predicate, a list of argument/result variables and their types.

[2] The term *site* is used to refer data sources and AMOS*II* servers. The terms *site assignment* and *node placement* are used interchangeably.

The grouping of the graph nodes is performed by a series of *node fusion* operations that fuse two nodes into one. The new node represents the conjunction of the predicates in the fused nodes and is connected to the rest of the query graph by the union of the edges of the fused nodes. MIF nodes are fused only with other MIF nodes belonging to the same DST capability set. Furthermore, the DST of the MIF nodes must have at least an SC join capability for a fusion to be applicable. The SIF nodes are fused only with SIF nodes to be executed at the same site, given that the following conditions are satisfied, on the basis of the site's join capability:

- *Site without join capability:* Nodes assigned this type of site are not fused. Each predicate is sent separately to the wrapper for processing. Typecheck predicates for the involved variables are added to aid the translation process in the wrapper.
- *Single collection joins site:* Two nodes are fused if they represent operations over the same collection in the source, represented by a proxy type in the query.
- *General join site:* Two connected SIF nodes, assigned to this type of a site, are always fused.

Assuming a query graph $G =< \mathcal{N}, \mathcal{E} >$, where $\mathcal{N} = \{n_1 \dots n_k\}$ is a set of nodes, and $\mathcal{E} = \{(n_1, n_2) : n_1, n_2 \in \mathcal{N}\}$ is a set of the edges between the nodes, the predicate grouping algorithm can be specified as follows:

while $\exists\,(n_i, n_k) \in \mathcal{E} : n_i$ *and* n_k *satisfy the fusion conditions* **do**
$n_{ik} := fuse(n_i, n_k)$;
$\mathcal{E} := \mathcal{E} - \{(n_i, n_k)\}$
$\mathcal{E} := \mathcal{E} \cup \{(n_{ik}, n_m) : (\exists (n_l, n_m) \in \mathcal{E} : n_l = n_i \ \vee \ n_l = n_k) \vee$
$(\exists (n_m, n_l) \in \mathcal{E} : n_l = n_i \ \vee \ n_l = n_k)\}$;
$\mathcal{E} := \mathcal{E} - \{(n_i, n_m)\} - \{(n_m, n_i)\} - \{(n_k, n_m)\} - \{(n_m, n_k)\}$;
$\mathcal{N} := \mathcal{N} - \{n_i, n_k\} \cup \{n_{ik}\}$;
end while

The algorithm terminates when all the possible node fusions are performed. After each fusion, the fused nodes are replaced in the graph by a new node, and all the edges to the fused nodes are replaced by edges to the new node. The $fuse$ function conjuncts the node predicates and adjusts the other runtime information stored in each of the nodes (e.g. typing and variable information).

After performing all the possible fusion operations the query graph contains nodes representing predicates that are to be submitted to the data sources as a whole. However, this is not the final grouping. The grouping is performed again after the MIF nodes are assigned sites (to be discussed below). Note that MIF nodes of different DSTs are not grouped together at this stage. Also, at this stage all the graph nodes contain either only MIF predicates or only SIF predicates.

The following example, used as a running example through the rest of the chapter, illustrates the grouping process. The query below contains a join and a selection over the type A in the source $DB1$, and B in the source $DB2$:

```
select res(A)
from  A@DB1 a, B@DB2 b
where fa(a) + 1 < 60 and
fa(a) < fb(b);
```

Two functions are executed over the instances of these types: $fa_{A\rightarrow int}()$ in DB1, and $fb_{B\rightarrow int}()$ in DB2. The calculus generated for this query is:

$$
\begin{aligned}
&\{\ r \mid \\
&\quad a = A_{nil\rightarrow A}() \wedge \\
&\quad b = B_{nil\rightarrow B}() \wedge \\
&\quad va = fa(a) \wedge \\
&\quad vb = fb(b) \wedge \\
&\quad va1 = plus(va, 1) \wedge \\
&\quad va1 < 60 \wedge \\
&\quad va < vb \wedge \\
&\quad r = res(va)\ \}
\end{aligned}
$$

The example query is issued in a AMOS*II* mediator and is over data stored in the data sources *DB1* and *DB2*. In the example, we will assume that these two sources have *Join* capability (e.g. relational databases or AMOS*II* servers). The initial query graph, shown in Figure 6.4a, has one node for each of the query predicates. The nodes are numbered with the rank of the predicates in the above calculus expression. In the figure, the predicates are shown beside each of the nodes. The nodes are labeled with the assigned site, or with "MIF" if they represent MIF predicates. The edges among nodes are labeled with the variables that connect the nodes.

Figure 6.4b shows the result of the grouping phase. The nodes $n8$, $n1$ and $n3$ are all assigned to the site *DB1* and make a connected subgraph, therefore they are fused into a node with the composed predicate:

$$a = A_{nil \to A}() \ \wedge \ va = fa(a) \ \wedge \ r = res(va)$$

The same applies for $n4$ and $n2$ at *DB2*. Although, $n5$ and $n6$ are both MIF nodes, they cannot be fused because they are of different DSTs: *arithmetic* and *comparison*, respectively.

6.1.3 MIF predicates execution site assignment

The nodes of the graph returned by the previous phase represent two types of predicates: SIF predicates that already have an assigned execution site, and MIF predicates that are still not assigned to a site. In order to generate subqueries for the individual data sources, the next step is to assign execution sites to the nodes containing MIF nodes. A MIF predicate can be executed at any site known to the mediator that is capable of performing the operations in the predicate. Furthermore, a predicate can be assigned more than one execution site, in which case it is replicated and executed at more than one data source. Because of the declarative nature of the predicates, this does not change the outcome of the query execution, as long as each predicate is executed at least once. Also, any assignment of an execution site to a MIF predicate yields a correct and executable query schedule; the difference is only in the costs of the generated plans.

Searching the space of possible site assignments using an exhaustive strategy would require examining every combination of known sites as execution sites for each of the MIF nodes. This would require performing full query optimization for each alternative using backtracking, resulting ultimately in an algorithm with exponential complexity. To avoid this expensive process, we tackle the MIF nodes site assignment problem by using a heuristic approach aided, in certain cases, with partial cost calculations.

The heuristic used is based on an analysis of the execution costs affected by a placement of a MIF node at a site. These costs are:

- the cost of the execution of the MIF predicate in the node.
- the execution cost of the predicates already assigned to the site where the MIF node is assigned

- the intermediate results shipment cost.

The first cost varies due to different speeds of the sites in the network. The cost of the execution of other predicates can change when a MIF node is fused with a SIF node placed at the same site, because the MIF node can represent a selection condition that significantly reduces the subquery execution time in the data sources. Finally, this kind of a selection will also influence the size of the intermediate results.

In order to simplify the placement problem, we recognize several different subcases and in each examine only some of the costs given above. In each case, the following goals are pursued in the order they are listed:

1. Avoid introducing additional cross-site dependencies among the nodes, caused by argument variables of the placed node. These dependencies often lead to increased transfer of intermediate results among the sites.

2. Place each MIF node so that it can be combined with one or more SIF nodes, to reduce the cost of accessing the data sources and to reduce the intermediate results sizes.

3. Reduce the execution time for the MIF nodes.

4. When it is not possible to assign a site to a MIF node on the basis of the previous three criteria, if possible execute the predicate locally.

The placement algorithm does not attempt to satisfy these goals simultaneously, but rather tries to satisfy one at the time in the order they are listed above.

Site assignment is performed one MIF node at a time. The nodes with more specific DSTs (further from the root of the DST hierarchy) are processed before the nodes with less specific DSTs (closer to the root of the DST hierarchy). For example, a MIF node with a predicate containing relational MIF operators will be placed before a node containing comparison predicates. The more specific DST nodes are always assigned to sites that can also process less specific DST nodes. Hence, a more specific node is always assigned to a node that also is considered when a less specific node is assigned. This is not true in the opposite direction, because the less specific node might be assigned to a site that does not have the capability to process the more specific node. Therefore, to maximize the possible available information at the node assignment time, the sources with more specific DST are processed first.

After each site assignment, the grouping algorithm is run over the new graph in order to group the newly assigned node with the nodes already assigned to the chosen site.

The site assignment process proceeds as follows. First, each calculus variable that labels an edge in the graph is assigned a set of sites where it appears, i.e. a set of the sites of the nodes that are connected by a graph edge labeled with this variable. This set is referred to as *variable site set*. Next, each of MIF nodes is processed. For each node, first an intersection of the site sets of the node's arguments is computed. This intersection represents the sites that operate over the same data items as the MIF node.

Figure 6.3 shows five subcases of the placement problem, distinguished by the properties of the argument's site sets intersection and the node predicate. The rest of this section examines each of the cases in greater detail.

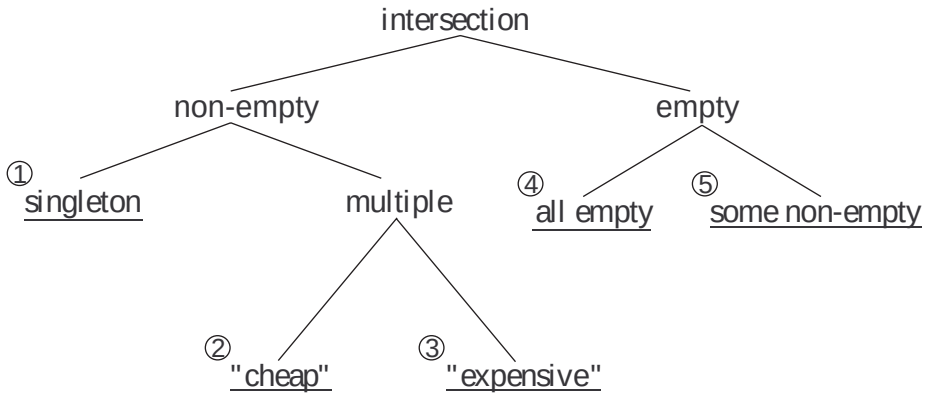


Figure 6.3: MIF predicate site assignment heuristics

Case 1: Singleton site sets intersection

If the intersection is not empty and contains only one site, then the node is assigned to this site. This allows the optimizer to devise a strategy where no intermediate result is shipped around when the node predicate is executed. All the arguments values can be produced locally at the chosen site. Placing the node predicate at a site where only a subset of the needed arguments can

be produced implies that the missing arguments must be shipped in before these predicates are executed. An example of this case of node placement is shown in Figure 6.4b where node 6 is connected only by the variable *va* to node 831. This node is assigned to the same site as 831, i. e. *DB1*. After the grouping of the graph the result is as presented in Figure 6.4c.

Cases 2 and 3: Several sites in the site sets intersection

The MIF nodes belonging to this case are placed on the basis of their cost and selectivity. If such a node has a selectivity lower than 0.75, and a "low" cost, then the node is considered to represent a cheap selection. The node predicate is therefore replicated, placing one copy at each of the sites in the intersection. The cost is considered low if it is lower than a predetermined constant threshold. The selective properties of the predicate are applied in multiple data sources. This strategy is unique to query processing in autonomous environments. In a classical distributed database environment, it would suffice to execute the selection at only one site. The query processor could then ship the intermediate result to the other sites, and use this already reduced set of instances as the inner set in the joins. When data sources do not support materialization of intermediate results, this strategy is not possible. Therefore, the selections should be pushed in all the applicable data sources to reduce the processing times in the sources, as well as proxy object generation in the translators associated with these sources.

Case 4: All site sets empty

A variable has an empty site set if it appears only in predicates of MIF nodes that have not yet been placed. If all site sets of the node arguments are empty, assuming a connected query graph, we can conclude that all the neighbors of this node are also unplaced MIF nodes. In order to obtain more information for the placement of such nodes, the decision on the placement of such nodes is postponed and the node is skipped. The skipped nodes are processed after processing the rest of the nodes. If all MIF nodes have all argument site sets empty, the first node is placed locally if possible. Otherwise, it is placed at the site where it will be executed fastest, i. e. at the most powerful site.

Assuming, that the site assignment proceeds in the same order as the nodes are numbered, in the situation shown in Figure 6.4b the algorithm will attempt to place n_5. Since n_5 is connected to only MIF nodes, its argument site sets intersection is empty. Thus, n_5 is skipped as described above, and

considered again when the rest of the MIF nodes are placed. The graph at this point is presented in Figure 6.4d. Now, the site set of variable *va1* is $Aset_{va1} = \{DB1\}$ since n_5 is connected to n_{8316}, placed at *DB1*, by an edge labeled *va1*. Node n_5 is therefore placed at *DB1*. After the grouping, the final query graph is shown in Figure 6.4e.

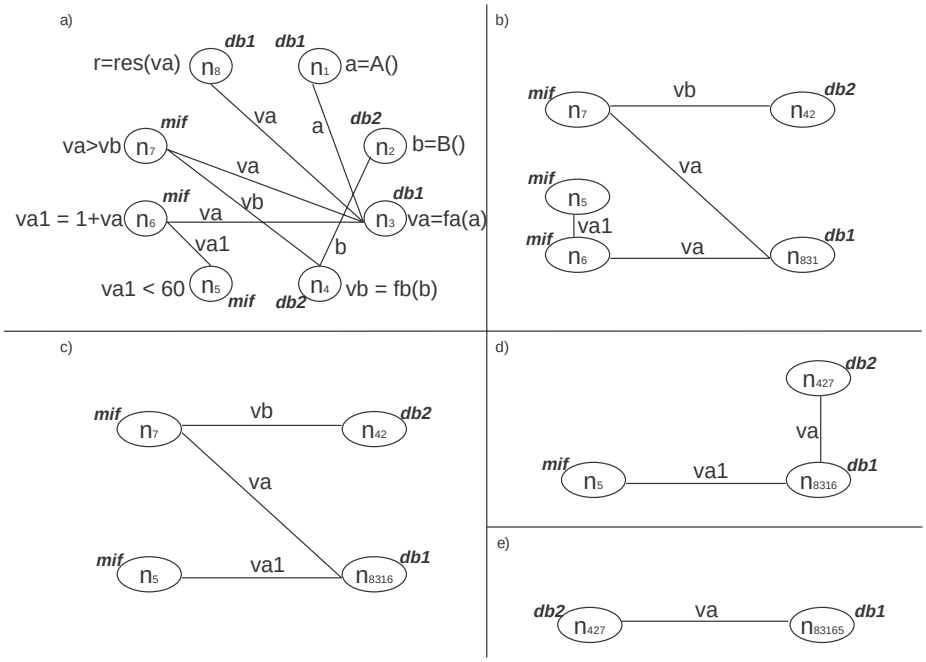


Figure 6.4: Query graph grouping sequence for the example query

Case 5: Non-empty site sets with empty intersection

In the last case, we consider placing a node having an empty intersection of its arguments' site sets, but not all of the site sets are empty. The placement process in this case is based on a simplified cost estimate. The estimate calculation takes into account only the predicates in the neighboring nodes of the currently processed node (this set coincides with the union of the arguments' site sets). Moreover, the cost estimate is calculated by taking

into account only the graph edges of the currently processed node. Another simplification of the problem is that nodes of this type are placed at exactly one site. Since no site contains all the data needed for the execution of the node predicate, the missing data must be shipped to the execution site from other sites. By placing the node at one site, we avoid plans having more than one data shipment caused by a MIF predicate.

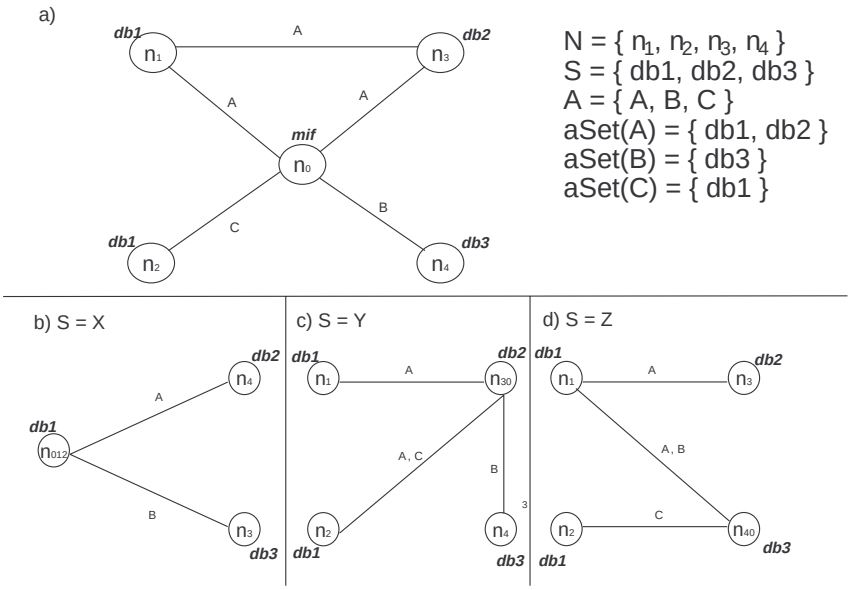


Figure 6.5: Case 5 example and the possible outcomes

For each of the possible placements, the sum of the execution costs of the predicates in the neighboring nodes and the necessary data shipment is calculated. The predicate is placed at the site where this cost is the lowest. Let the list of the neighboring nodes of a MIF node labeled with the node site be $N = \{n_1^{s_{n_1}}, \ \ldots \ , n_l^{s_{n_l}}\}$; the sites the nodes are assigned to $S = \{s_1, \ \ldots \ , s_m \}$, $m \leq l$; the node predicate argument variables $A = \{a_1, \ \ldots \ , a_k\}$; and finally, the corresponding site set to each variable: $As = \{aSet_1 \ \ldots \ , aSet_k\}$.

The execution cost of the nodes at site s if each predicate is executed

over BS (bulk size) tuples is defined as the sum of the costs of the individual nodes:

$$exec\_cost(s, BS, A) = \sum_{j=1...l, s=s_j} cost(n_j^{s_{n_j}}, BS, A)$$

Where the *cost* function returns the cost of executing the predicate in a given node with the arguments in A unbound. In calculating the estimate, the number of input tuples is fixed to a predetermined constant, because it cannot be precisely estimated before the scheduling phase, and varies for different nodes. Using a constant value for each estimate provides a good basis for comparison of the estimates. However, it is important that this constant is larger than 1 in order to correctly estimate the effect of the use of subquery materialization techniques in queries containing nested subqueries. In such cases, the query processor might decide to materialize the subquery and use the result in the processing of the whole input. The cost of the materialization is amortized over the processing of the whole input and therefore:

$$cost(n_j^{s_{n_j}}, BS, A) \neq BS \cdot cost(n_j^{s_{n_j}}, 1, A)$$

Nested subqueries are common in the system-generated functions for support of the integration union types presented in the previous chapter, making this type of cost estimate necessary.

When a node is placed, the grouping algorithm is applied to the new subgraph. The sum of the costs of the nodes in the grouped subgraph is denoted with $pa\_exec\_cost(s, BS, vl)$. Assuming that the node is assigned to a site S where a subset of A_l of the argument set A is produced locally while the rest of the arguments $A_t = A - A_l$ are shipped from the neighboring nodes, the execution cost estimate can be expressed as:

$$ece(s) = pa\_exec\_cost(s, BS, A_t) + \sum_{i=1...l, s_i \neq s} exec\_cost(s_i, BS, A_l)$$

To obtain a complete cost estimate, besides the execution cost estimate, we need to compute an estimate for the intermediate results shipment cost. To calculate this estimate we assume that each of the missing arguments in A_t is shipped to the site S from the cheapest possible alternative. The cost of shipping the argument $a_i \in A_t$ from a site R where it is produced by the predicate of node N to site S where it is consumed is:

$$tec(a_i, N, S) = BS \cdot selectivity(N, A_t) \cdot W_{RS} \cdot sizeof(type(a_i))$$

Where W_{RS} is the weight of the cost of the net link between the sites R and S; $selectivity(N, A_t)$ returns the selectivity of the predicate of node N with all arguments in A_t unbound; $sizeof()$ returns a size of a given tuple of types; and $type()$ returns a tuple of types for a given tuple of variables.

$$tec(S) = \sum_{a_i \in A_t} \min_{n_i \in N} tec(a_i, N, S)$$

The cost estimate is:

$$ce(s) = ece(s) + tec(s)$$

The node is assigned to the site so such that

$$\forall n \in N \; ce(so) \leq ce(n)$$

Although all the possible site assignments produce a correct execution plans, the cost estimate calculation can fail for some sites, because some of the functions might not be executable with the incomplete binding patterns used to calculate the estimate. Such sites are ignored in the assignment process. In a rare case, it is possible that all the estimate computations fail. In this case, an arbitrary site is chosen from the set of sites capable of handling the node predicate.

In order to estimate the complexity of the cost estimate calculation we can observe that the terms used in the equations above can all be obtained either from the system catalogue (e. g. $sizeof()$ function and W_{RS}), or from compilation of the predicates in the query graph nodes (the $cost()$ and $selectivity()$ functions). The maximum number of compilations needed to obtain this data is $2l$, where l is the number of adjacent graph nodes of the node being placed. This estimate is based on the observation that each neighboring node predicate is compiled twice: once for the case when the node is placed at the same site with the neighboring node, and once when it is placed at another site. Normally, the queries posed to the mediator have connected query graphs, implying that $l \leq n$, n being the number of sites involved in the query. Hence, the cost of the site assignment will usually not be larger than $2n$ single site function compilations, some of which might be reused in the latter decomposition phases. We also note that n here does not represent all sites involved in the query, but rather the sites that operate over the arguments of the predicate in the placed node.

In Figure 6.4c the node n_7 represents an example of case 5 placement problem. The example illustrates the problem of the placement of the join

condition $va < vb$. The cost estimation will ignore the node n_5 and will calculate the costs as described above. Figure 6.4d shows the graph after placing n_7 at $DB2$.

A more elaborate example of this case is illustrated by the query graph shown in Figure 6.5a. On the right side of the Figure the sets used in the calculations of the estimate are shown. There are three sites involved with a total of 4 nodes. Assuming *Join* capabilities, the resulting grouped graphs for each placement alternative are shown in the Figures 6.5b-d.

This concludes the description of the query decomposition phases that assemble the subqueries sent to the individual data sources. The concepts discussed in the previous sections are related to the important design issue of the division of the query processing facilities between the query decomposer and the wrappers. A simple query decomposer requires more complex wrapper implementations. A wrapper in such a case must be able to perform more sophisticated transformations in order to produce subqueries executable by the data sources. Furthermore, the same features might be needed and reimplemented in several wrappers. A more elaborate query decomposer, on the other hand, leads to a slower query decomposition and less maintainable code. The design of the heterogeneous data source integration facilities described in the last two sections aims to provide a functionality sufficient for easy integration of the majority of the data sources we have accounted for, while keeping the design as simple as possible. Compared to other approaches to the integration of heterogeneous data sources based on grammars and rules [36, 81], it allows for partitioning of the query into subqueries without repeated probing if the generated subqueries are executable in the data sources. Data sources that cannot be described by MIFs and join capabilities might require wrappers capable of restructuring the subquery sent by the decomposer so it can be successfully translated into code executable in the data sources. Nevertheless, we believe such that cases are rare.

6.1.4 Cost-based scheduling

The result of the first two query decomposition phases is a query graph where each node represents a subquery assigned to be executed at a site. The graph nodes are connected by edges representing equi-joins over the values of common variables in the subqueries. In order to translate this query graph into an executable query plan, the query processor must decide on the order of the execution of the subqueries represented by the nodes.

This order influences the data flow between the sites. The query processor builds an *execution schedule* to describe the execution order and the flow of data between the sites.

As noted earlier, the data in each node is used to define a derived function representing the subquery specified by the node predicate. These derived functions, *subquery functions* (SFs), are defined at the site assigned to the corresponding node when this site is an AMOS*II* server, or in the mediator itself when the SF is executed in the mediator or in a data source wrapped by the mediator. In the later case, the SF is generated by the wrapper, invoked with the node predicate as an argument. The wrapper returns a function that implements the request specified by the input predicate. The generated functions usually contain foreign function calls that access the data source and perform the requested operations. For example, the relational wrapper implemented within the AMOS*II* project creates an SQL statement from the object calculus, and then invokes the foreign function *sql* [6] that passes the generated SQL statement to a data source.

Examining all the possible execution schedules is not feasible for larger queries. Considering only the left-deep trees (a subset of all possible trees) is as hard as finding an optimal total ordering of the predicates. Although simplified, this problem still requires computation time exponential over the number of SFs. Therefore, only certain *schedule families* are examined that contain plans generated by a few generic rules.

The scheduling problem is illustrated on the running example query. The final query graph for this query contains two nodes, each specifying an SF at one of the two participating sites. The definition of the SFs at these nodes are as follows:

in DB1:

$$SFdb1_{type\_va \rightarrow boolean}(va) \Longleftrightarrow$$
$$\{$$
$$b = B_{nil \rightarrow B}() \wedge$$
$$vb = fb(b) \wedge$$
$$va < vb\}$$

in DB2:

$$SFdb2_{type\_r,type\_va \rightarrow boolean}(r, va) \Longleftrightarrow$$
$$\{$$
$$a = A_{nil \rightarrow A}() \wedge$$
$$va = fa(a) \wedge$$
$$va1 = plus(va, 1) \wedge$$
$$va1 < 60 \wedge$$
$$r = res(va)\}$$

The function signatures used above imply that both SFs will be executed with all their arguments bound. Such binding patterns are used in the SF definitions, because the binding pattern of each SF is unknown at this time

and is determined later in the scheduling process, by recompilation of the SFs.

Let's consider now the possible execution strategies for the example query. The query execution begins by executing one of the SFs at one of the sites. Then, the other SF is executed and the result is shipped to a join and materialization capable site, where an equi-join over the variable va is performed. This site could be one of the sites where the SFs are defined. In such a case, we could use the materialized result as an input to the SF at this site, to lower the execution time and the selectivity of the individual predicates in the body of the SF. For example, if $SFdb1$ is executed first and the resulting va values are shipped to $DB2$, we could either first execute the function $SFdb2$, and match the resulting tuples with the materialized values of va, or invoke $SFdb2$ with the values of the argument va bound to the values in the shipped set. In order to determine the optimal schedule, the query processor must calculate and compare the costs of the different strategies. The cost calculation depends on the execution cost and the selectivity of the SFs and the cost of shipping data among the systems.

This analysis illustrates that the number of alternatives is large even in a simple example with only two SFs as above. Because of this, the strategy described in this section searches only a portion of the search space of possible execution plans. The plan chosen by this search is then improved using additional heuristic described in the next section.

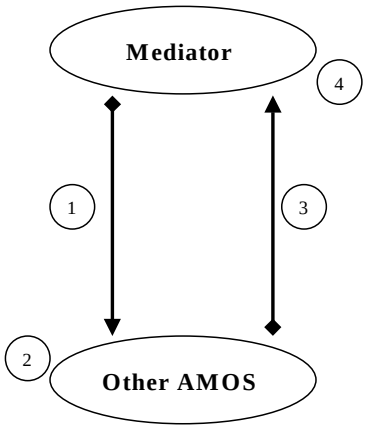


Figure 6.6: A query processing cycle described by a DcT node

The generated execution schedules are described in the form of *decomposition trees* (DcTs). Each DcT node describes one data cycle through the mediator. Figure 6.6 illustrates one such cycle. In a cycle, the following steps are performed:

1. Materialize the intermediate results in an AMOS*II* server where they are to be processed.
2. Execute an SF over the materialized data as input.
3. Ship the results back to the mediator.
4. Execute one or more SFs defined in the mediator.

The result of a cycle is always materialized in the mediator. A sequence of cycles can represent an arbitrary execution plan. Not all steps are required in every DcT node.

Each DcT node contains data structures describing the steps above. The intermediate results used as an input in the cycle are represented recursively by a list of child DcT nodes, the *materialization list*. In order to simplify the query processing, currently the tree building algorithm considers at this stage only materialization lists with one element (left-deep trees), and therefore the intermediate result always has the form of a single flattened function.

Steps 1 through 3, that involve communication with an another AMOS*II* server, are performed by the *ship and execute* (SAE) operator. The SAE operator is an algebraic operator that ships an intermediate result to a remote AMOS*II* server, executes an SF, and returns the result. Each tree node contains an *SAE description structure* (SAEDS) that provides the necessary compile-time information about the ship and execute performed by the node. The content of an SAEDS describe the remote SF and the way it is invoked. More specifically this description consists of the following items:

- proxy OID for the remote SF
- argument and result lists
- argument bindings and typing information
- cost and selectivity of the SF for a given binding

Step 4 is described by a *post-processing list* (PPL) of locally defined SFs. These SFs are executed in the mediator over the result of the SAE operator

execution. Finally, beside a materialization list, a SAEDS and a PPL, each DcT node also contains information concerning the whole query processing cycle described by the node, as for example: cycle cost, selectivity, predicates, typing information, etc.

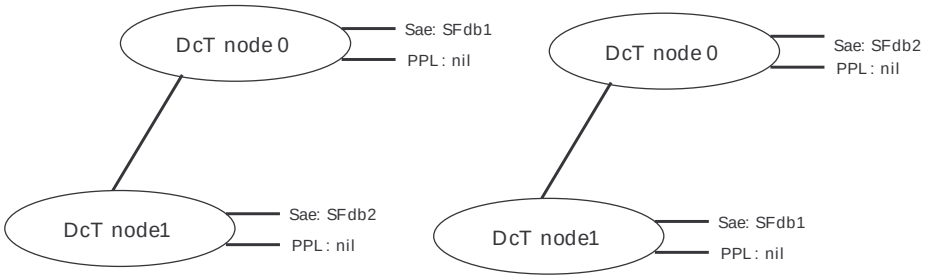


Figure 6.7: Two decomposition trees for the example query

Figure 6.7 shows the two trees generated for the example query. These trees illustrate the scheduling alternatives where the join of the results of the execution of the two SFs is performed at $DB1$ and $DB2$ respectively. Because we consider only left-deep trees, joins in the mediator are not considered. The trees also determine the relative order of the execution of the SFs. The order of the cycle operations given above implies that the trees are executed bottom-up. This in turn determines the execution binding pattern for each SF. The same SF can in different trees have different binding patterns and thus different execution costs. In the left DcT in Figure 6.7 $SFdb2$ is executed with the variable va unbound, while in the tree on the right this variable is bound. If the function $fa(a)$ is expensive, or has high selectivity, then the execution of $SFdb2$ with va unbound can have a much higher cost than when va is bound. This cost variation combined with the cost variation of $SFdb1$ influences the cost of the whole tree.

The cost of an execution schedule represented by a DcT node is calculated recursively by adding the costs of the steps in Figure 6.6 to the costs of the subtrees in the materialization list. The cost calculation depends on the algorithms used to implement the query processing cycle steps. These algorithms are part of the query execution mechanism described in the next section.

The left-deep DcTs are generated using a variation of the dynamic programming approach. The algorithm attempts to avoid generation of all the possible plans by keeping a sorted list of partial plans and adding to the list all the possible extensions of the cheapest one. When the cheapest plan is also a complete plan, then it is one of the plans with the lowest cost. This algorithm, used also for the singe-site queries in AMOS*II*, can be described as follows:

```
find_optimal_schedule(SF_set)
   /*sorted list of partial DcTs*/
   list<DcT> DcT_list = {};
   set<DcT> rest;
   /* temporary variables for DcT manipulation*/
   DcT best, nd;
   for each func in SF_set
     nd = add_to_DcT(func, nil);
     insert_sorted(nd, DcT_list);
   end for each
   forever do
     best = remove_top(DcT_list);
     rest = SF_set - DcT_SF(best);
     if best == nil
       throw exception(``Query unexecutable'');
     end if
     if rest == {}
       return best;
     end if
     foreach func in rest
       nd = add_to_DcT(func, best);
       insert_sorted(nd, DcT_list);
     end_foreach
   end forever
end
```

If the query is not executable an exceptions is thrown The function *insert_sorted* inserts a DcT in a list sorted by the cost estimate; *remove_top* removes the cheapest plan from the list; *DcT_SF* returns all the SFs in a DcT; the operator − is used for set difference; the function *add_to_DcT* adds a new SF to a partial DcT. The following two rules are used for adding an SF to a partial DcT (Figure 6.8):

- **SF defined in another AMOS*II* server**: A new node is added with a materialization list consisting of the partial DcT, and a SAEDS based on the added SF.
- **SF defined in the mediator or in a data source wrapped by the mediator**: The SF is added to the PPL of the root of the DcT, if the DcT is not *nil*; otherwise a new node is created with this SF in the PPL.

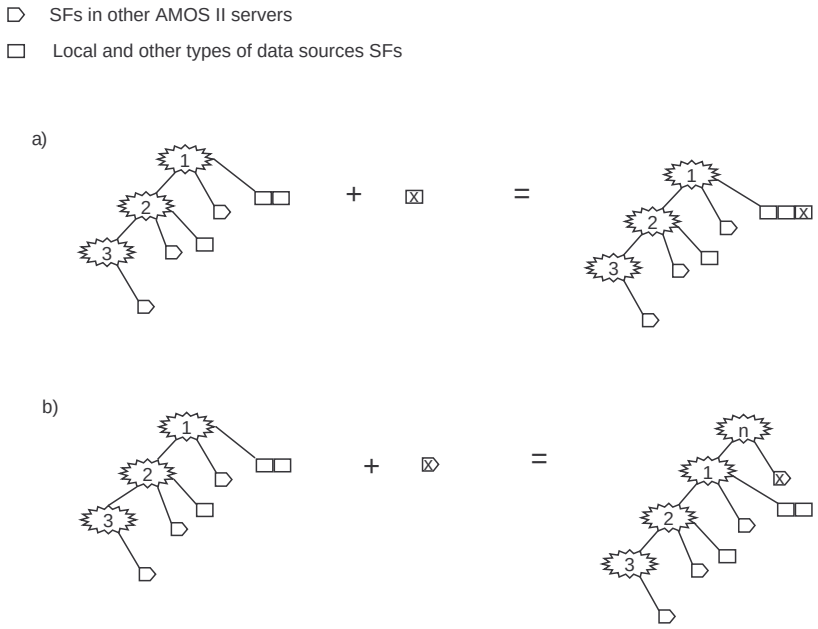


Figure 6.8: Two tree generation rules: a) adding a local SF to a partial tree, b) adding a remote SF to a partial tree

When an SF is added to the PPL of a node, the system must determine the optimal order of the execution of the SFs in the list. This cost influences the cost of the whole tree and therefore must be determined during the query optimization. A dynamic programming algorithm similar to the on described above is used to determine an optimal ordering.

We conclude the section with an observation that the described strategy is more general given OO data sources than the strategies used in some other multidatabase systems (e.g. [56, 20, 48]) where the joins are performed in the mediator system. Such strategies do not allow for mediation of OO sources that provide functions that are not stored, but rather performed by programs executed in the data source (e.g. image analysis, matrix operations). In this case, it is necessary to ship intermediate results to the source in order to execute the programs using the result tuples as an input. From this aspect, the strategy presented above generalizes and improves the *bind-join* strategy in [36].

6.1.5 Decomposition tree distribution

The scheduling phase described in the previous subsection produces a left-deep DcT representing a query execution schedule for the input query. As described in the previous section, each node of the generated DcT describes a query processing step that involves passing data through the mediator. Some of the steps pass data from one data source to another, copying it through the mediator. In an environment consisting of a several AMOS*II* servers, it can be favorable to design schedules where the superfluous data transfers and the involvement of the coordinating mediator are eliminated. In such a schedule, the participating data sources communicate directly during the computation phase. The result of the computation is shipped to the coordinating mediator. For example, the trees in Figure 6.7 describe plans in which the values of *va* are shipped from $DB2$ to the mediator and then to $DB1$, in the tree on the left, and vice versa in the tree on the right. It would be less costly if the mediator instructs $DB2$ to ship the values directly to $DB1$, or vice-versa.

In this subsection we present a technique to achieve query schedules with such properties for queries operating over data represented by multiple AMOS*II* servers. The technique can be extended to combine other materialization capable DST.

In order to construct schedules that perform "sidewise" transfer of data, the DcT generated by the previous phase is restructured using a series of *node merge* operations. This operation is performed over two consecutive nodes, *lower* and *upper* respectively. For the merge operation to be applicable, the lower node must have an empty PPL, while the upper node must have a non-empty SAEDS. The tree construction rules guarantee that a node having

an empty PPL must have a non-empty SAEDS. Therefore, the lower node describes an operation where data is shipped from the mediator to another AMOS*II* server, some computation is performed there, and the result is then shipped back to the mediator. In the next step, described by the upper node, the result of the previous step, stored now in the mediator, is once again shipped to the site described in SAEDS of the upper node, this time as input to the SF in the upper node SAEDS. For example, the left tree in Figure 6.7 describes a plan where $SFdb2$ is executed at $DB2$ and the result is shipped to the mediator. The upper node then ships the same result from the mediator further to $DB1$, where it is used in an equi-join.

In the case when the lower node has a non-empty PPL list, the result of the lower node is processed locally before it is used in the processing step described by the upper node. The transformations described in this section are, therefore, in such a case not applicable.

The node merge operation is shown in figure 6.9. Two consecutive nodes with the required properties are identified (Figure 6.9a) and substituted with a single node. The new node has the PPL from the upper node and a SAEDS assembled from the SAEDSs of the merged nodes. In order for the new tree to represent a correct query schedule, the SF in the SAEDS of the new node should perform the same operations as both SAEDSs of the original nodes. Therefore, the SF in the new node's SAEDS is a combination of the SFs of the original nodes. Nevertheless, to avoid the unnecessary bypass of the data throughout the mediator, the combined SF is compiled and executed in the participating servers instead of locally in the mediator. This is done by defining an *envelope SF* that calls the two original SFs. The envelope SF is compiled at both of the participating sites and the cheaper alternative is accepted. This, in turn, is compared with the cost of the original tree and if it has a lower cost, the modified tree is accepted instead of the original.

Figure 6.10 illustrates the data flow between the three AMOS*II* servers in the example from Figure 6.7a. This example is a general case of an application of a node merge. An exception is the case where the merged nodes have SAE SFs that are both at the same site. In such cases there would be only two sites involved.

In Figure 6.10a the data flow of the execution of the original schedule is presented. The query execution starts by the mediator contacting $DB2$ to execute $SFdb2$ in step 1, and shipping across the results in step 2. Next, from the result of the previous step, the mediator sends the values of the argument va to $DB1$ where $SFdb1$ is executed and the result is joined with

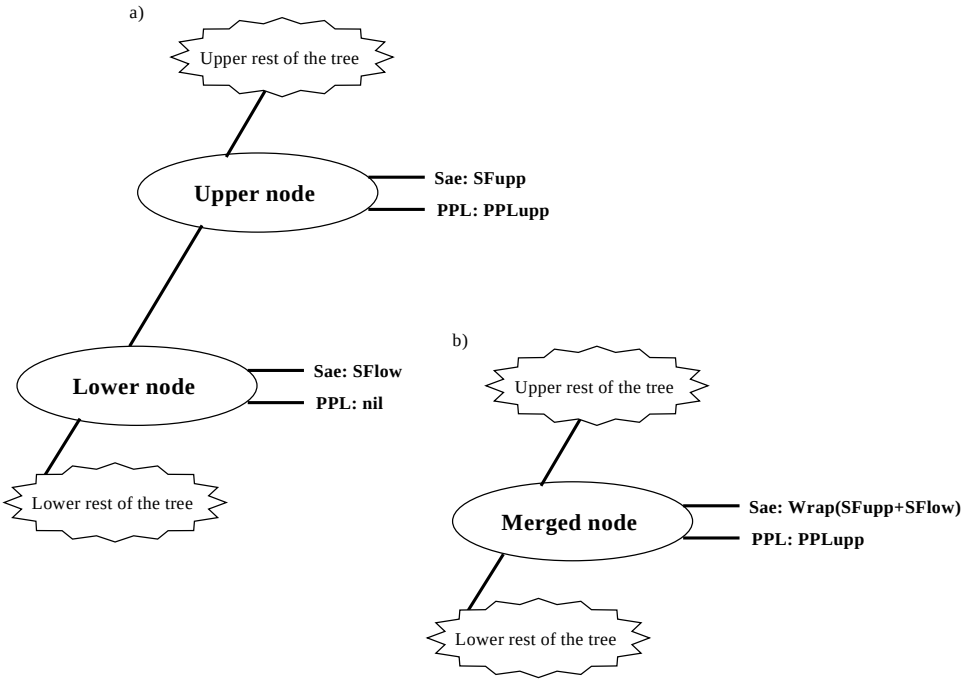


Figure 6.9: Node merge: a) the original tree b) the result of the merger operation

the incoming set of va values. For each joined value of va a temporary boolean value is returned indicating which of the incoming va values joined with the result of the execution of $SFdb2$. Finally, after joining the result shipped in step 4 with the result of step 2, the mediator emits the values of r for which the temporary iteration variable $\_tmp\_$ is $TRUE$.

This strategy would be very inefficient in cases when the set of va values is very large and the net links connecting the mediator with $DB1$ and $DB2$ are very slow (e.g. due to geographical dislocation). Also, note that with this strategy the va values are shipped twice.

The strategy illustrated in Figure 6.10b is obtained by merging the nodes of the DcT in Figure 6.7a and placing the envelope SF at $DB2$. Here, the values of va are sent directly from $DB2$ to $DB1$, shipping them therefore only once. Figure 6.10c represents the execution strategy of the transformed

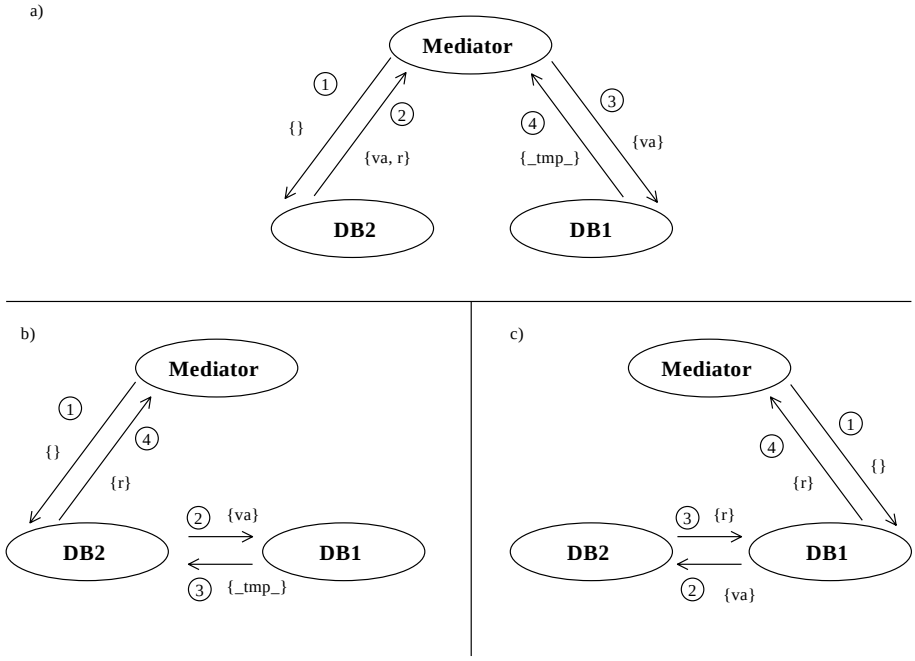


Figure 6.10: Execution diagrams of the decomposition tree of the example query before node merge and after

DcT in Figure 6.7a where the envelope SF function is placed at $DB1$. This strategy is favorable when $SFdb1$ has large selectivity or the network link between the mediator and $DB2$ is slow.

A series of node merge operations can produce longer data flow patterns that do not necessarily pass through the query issuing mediator. One feature of the trees produced by node mergers is that the SFs in the SAEDSs are themselves multidatabase functions over data in multiple data sources. These SFs are also compiled and described by a DcT. Since by repeated application of the merging process their SAEDSs can also have SFs over multiple data sources, we can assume that a query is represented by a *set of DcTs* distributed over more than one AMOS*II* server. Hence, the process can be viewed as *DcT distribution*. Compared with the traditional query tree balancing [17] the node merge exhibits the following differences:

- Distributed compilation: node merging is a distributed process where the envelope SFs are compiled at nodes other than the mediator. This distributed compilation process is decentralized and does not need a centralized catalogue of optimization information that is a potential bottleneck when the number of mediators increases.

- Distributed tree: The resulting tree is not stored in one AMOS*II* server, but rather is spread over the participating servers that expose only an already compiled function for the subquery sent by the coordinating mediator.

In a tree produced by the cost-based scheduling there might be more than one spot that qualifies for a merger operation. An important issue in applying node merging is where in the DcT to apply the the operator. Different sequences of merge operations can produce different results. The simplest solution to this problem is to perform an exhaustive application of all possible sequences of merger operations by backtracking. However, it is clear that this will require a large number of SF compilations and is therefore not suitable. An alternative is to use hill-climbing from a few randomly chosen positions and perform the process until no transformation can be made such that a cheaper tree is produced. The process can be guided by heuristics that prioritize DcT nodes where the transformation can be especially useful, and avoid merging nodes that are unlikely to produce a merged node with lower cost. An example of such heuristic rules are:

- Merge only nodes where the SF in the SAEDS of the lower node has at least one result variable used in the input of the SF in the SAEDS of the upper node. This rule avoids producing merged nodes where the result shipped to the mediator is a cross product of the results of the two SFs.

- Merge nodes where the weight of the network connection between the SAE SFs in the upper and the lower nodes is considerably lower than the weights of their network connections with the mediator (e.g. the data sources are close to each other, but geographically far away from the mediator)

6.2 Object algebra generation and run-time support

6.2.1 Object algebra generation

The purpose of the DcT formalism is to provide a means for representation of query execution schedules that is easy to build and manipulate. It contains much information not used after the tree is built. In addition, the components needed for execution are not easily and uniformly accessible.

For these reasons and to avoid introducing a separate interpreter for the DcTs in the system, the DcTs are translated into object algebra plans used for the queries over local data. The algebra plan generated for each DcT node describes the tasks specified by a node:

1. Execute and materialize the subqueries described by the DcTs in the materialization list.
2. Ship the results and execute the SF in the SAEDS.
3. Execute the SFs in the PPL.

The translation generates a temporary function whose body describes the listed tasks, returning the query result. The algebra code for each of the DcT nodes is generated using the following template:

1. $makebag^{bbb\ldots f}([subq\_func], [par_1], [par_2], \ldots [par_n], bag_1)$
2. $bulken^{bf}(bag_1, bag_2, [bulk\_size])$
3. $sae^{bbf\ldots f}([run\_info], bag_2, [\texttt{SAE result variables}])$
4. $PPLSF_1($ $[pplsf_1$ `arguments`$)]$

...

k+3. $PPLSF_k($ $[pplsf_k$ `arguments`$])$

The arguments in square brackets are substituted by actual values during the code generation. The algebra operators used in the first three steps are implemented as an AMOS*II* foreign function. The superscripts of the operators show the binding pattern applied. Steps 4 through $k+3$ invoke the SFs in the PPL of the node.

In order to perform the materialization of the subtrees, the translation process is applied recursively on the DcTs in the materialization list. As currently, due to the tree generation process used, there is only one subtree in this list, the result of this step is a single temporary function storing the

result of the tree in the materialization list. This function is in the code template above named *subq_func*. It is the first argument of the operator *makebag* that takes a function and a list of parameters to the function, and returns a bag which can be iterated by the bag iteration primitives. The bag mechanism provides a uniform interface to both a materialized bag, given as a list of tuples, and a bag given with a function as in the case above. When the bag is specified by a function, the *makebag* operator does not actually materialize the function. Instead, when the iteration operations are applied to the bag, it uses the cursor facilities in AMOS*II* to invoke the supplied function passing the parameters $par_1 \ldots par_n$, and returning the function results in a streamed fashion. If the bag is constructed over (i) a stored AMOS*II* function, (ii) a foreign function whose implementation supports streaming, or (iii) data in a source whose wrapper supports streaming, then the execution of the *makebag* does not store intermediate results in the mediator. An example of a DST capable of a fully streamed operation is the *ODBC* DST, where, using the cursor facilities of the ODBC standard, it is possible to implement a wrapper where the tuples from the data source(s) are streamed through the AMOS*II* translators/mediators, to the target applications.

The bag produced by the *makebag* operator is assigned to the variable bag_1 in the code template above. In the next step, the *bulken* operator materializes the first *bulk_size* tuples of bag_1. This operator backtracks and produces materialized bulks of this size until the whole bag_1 is iterated.

Although the streamed mode of operation is advantageous because it does not create intermediate results, operating over a single tuple in a network-based environment can create large overheads. Therefore the multidatabase query execution in AMOS*II* is performed over bulks (sets) of data of a size determined by the parameter *bulk_size*. The parameter *bulk_size* is set by the optimizer to a constant depending on the available memory size, so that the intermediate results can fit into the available main memory image of AMOS*II*. When networks with high latency are used, the *bulk_size* parameter should be chosen so that the shipped data fits into the network's packet size and hence minimizes the latency overhead per shipped tuple. The bag containing the materialized tuples is assigned to the variable bag_2, passed to the SAE operator.

6.2.2 Inter AMOS*II* communication and the SAE operator

The SAE operator performs the actual communication between the AMOS*II* servers, the invocation of the remote subquery functions, and the assembly of the results. The protocol executed within the SAE operator is the lowest "database aware" protocol in the AMOS*II* system. It is based on a remote evaluation protocol that supports shipment and evaluation of LISP expressions between the servers. The LISP expressions are shipped using TCP/IP sockets connections between ports acquired from the name server.

The first argument to the SAE operator is the run_info structure that contains the run-time information needed for the execution of the SAE operator. Run_info is a summary of the SAEDS used during the query compilation time. Some of the more important entries in run_info are:

- The site of the remote SF: the AMOS*II* server where the SAE SF is defined
- The set of input function variables: description of the input bag structure by the query variable names of each bag column.
- The set of input variables to the remote SF: a projection of the input bag variables used at the remote site as arguments in the execution of the remote SF, *KS*.
- The set of result variables: variables that are to be emitted from the SAE operator and that are used in the subsequent query processing steps, *RS*.
- The bulk size, BS, and the estimated number of bulks in the input, NB.

The second SAE operator argument is the bag bag_2 materialized by the *bulken* operator.

The SAE operator is performed over each of the materialized subsets of the input function, represented by the bag bag_2. The tuples in the bag are used as input to the SF in the SAEDS, defined in some other AMOS*II* servers. The operations performed by the SAE operator can be divided into the following steps:

1. preprocess and prepare the input for shipping
2. ship the input to a remote site

3. execute the SF at the remote site
4. return the result of the SF execution to the coordinating mediator
5. assemble the result to be emitted from the operator

Steps 1, 4 and 5 are executed locally, while steps 2 and 3 are performed at another AMOS*II* server. The SAE operator returns values by binding the `SAE result variables` in the template above to the values in the tuples from the result of the execution. The result of the SAE operator is emitted a tuple at a time.

Figures 6.11 through 6.13 illustrate three algorithms for implementation of the SAE operator. All three implementations have two parts: the SAE operator in the coordinating mediator (the one initiating the query execution), and a request handler at the site where the SAE SF is executed. Before discussing the specifics of each of the algorithms, we first turn the attention to the problem of transferring OIDs and type information in a distributed architecture encompassing a number of autonomous OO mediator systems.

OIDs and types in the Inter AMOS*II* communication

Within a mediator, the OIDs of the objects from other AMOS*II* servers are represented as proxies. As described earlier, each proxy object is associated with an instance of the type *foreign_oid* that represents *stringified OIDs* from other servers. The stringification is a process of translating the OIDs into a sequence of bytes that can be transferred over the network. The stringified OIDs can be reversed to ordinary OIDs in the originating AMOS*II* server.

When a query over a proxy type or a type derived from a proxy type is executed, the generated schedule might require that some OIDs imported from another AMOS*II* server and stored locally are to be sent back to the originating server where an SF is performed over them. An example of such a query is the query in section 4.2.5 where the OIDs of type *Person* defined in the sport database are shipped back to evaluate the *hobby* function.

When the system detects that proxy type instances are shipped out of the mediator, it substitutes them with the associated stringified OIDs of type *foreign_oid*. The oval named *Deproxify OIDs* in Figure 6.11 describes this step.

Next, when a SAE request handler accepts data and an SF to be executed, it first performs an analysis of the incoming data. In the case where

there are stringified OIDs of some local type objects, it transforms each of them to a valid OID, if the object with that OID is still present in the database, or to NIL otherwise. This process is described by the oval named *Destringify OIDs* in the SAE request handler in the Figure 6.11.

The inter-server data shipment paths in a query execution schedule do not necessarily follow the same paths as the type importation. Hence, the data coming into a SAE request handler might contain columns with OIDs of types that are not known (imported) at the executing server. In such cases, the executed SF is a function over data in multiple data sources and the incoming data is passed unaffected to some of the servers participating in the SF execution. Stringified OIDs are passed unaffected in all intermediate AMOS*II* servers, until they arrive to the originating server. The originating server is the only site where operations can be performed over these OIDs. OIDs bear no semantics outside the server where they are created, but can be stored in systems that have directly imported their type from the originating server, and send back to the originating server for processing.

A query result of a query can also contain some imported type OIDs. When a query retrieves instances of a type imported from another AMOS*II* server (e.g. to be stored in local functions), the query decomposer might generate execution schedules where these are passed through intermediate servers that do not know this type. To be passed out of the originating server, the OIDs are stringified. They pass unaffected through the intermediate servers. On arrival at the target server, proxies are generated for the incoming stringified OIDs storing the incoming $foreign\_oid$ data in a stored function. In Figure 6.11 these two complementary steps are described with the *Stringify OIDs* and *Proxify OIDs* ovals.

The described protocol allows for transportation of typed information without following the conceptual connections between the mediators set by the type importations. It also allows for flexible data shipment paths in a distributed mediation environment without global schema. The protocol implementation is based on the types of the data shipped among the servers. The type information is needed to perform operations over OIDs in the shipped data. Because the subqueries that receive the shipped data are fully fledged AMOS*II* functions, they store the types of the input and the output in their signatures. Using the function signatures as a source for the typing and the tuple width information allows for a very lightweight data shipment protocol containing only the shipped data items.

SAE operator algorithms

Besides the described OID transformations, the SAE operator performs operations to combine the input data with the data obtained from the execution of the remote SF into a set of resulting tuples bound to the output variables. The input to a SAE operator can be viewed as a table consisting of columns labeled by calculus variables. Some of the columns are used as an input to the remote SF invoked by the SAE operator. Other columns are used in the query processing steps described by nodes above the current one in the query DcT.

A naive implementation of the SAE operator would ship the whole input bulk to the remote site, execute the SF appending its result to the input, and then ship this result back. Many redundancies can be noted in this approach. The first improvement of the naive strategy we propose is the *project-concat algorithm* (PCA). Its principal steps are shown in Figure 6.11. It avoids some of the redundancies of the naive strategy by the following two data transformations:

- The input bulk is projected over the data columns that are actually used in the remote SF, before shipping them there.
- After the SF is executed the result shipped back to the mediator contains only the relevant columns (i.e. columns used later in the processing or part of the query result) from the SF execution result.

The result of the SAE operator is assembled by a simple concatenation of the input and the result shipped back from the remote AMOS*II* mediator. Since the operations are order preserving, concatenation can be used instead of the more expensive join. In the case when the remote SF does not return relevant data, the result is a bulk of boolean values, one for each input tuple. The input tuples such that the result of the SF execution is NIL (no matching tupes) or $false$ in case of a boolean SF are deleted from the resulting set. Before emitting the result of the SAE operator, the concatenation of the input and SF result is projected over the set of columns relevant for the rest of the query execution (RS).

Table 6.1 presents an example execution of the PCA. The example simulates the PCA execution for a decomposition schedule of the running example introduced on page 96 in this chapter. The decomposition for this query is presented on page 106. The example is executed assuming the execution schedule shown in Figure 6.10 with $DB1$ being another AMOS*II* server. The

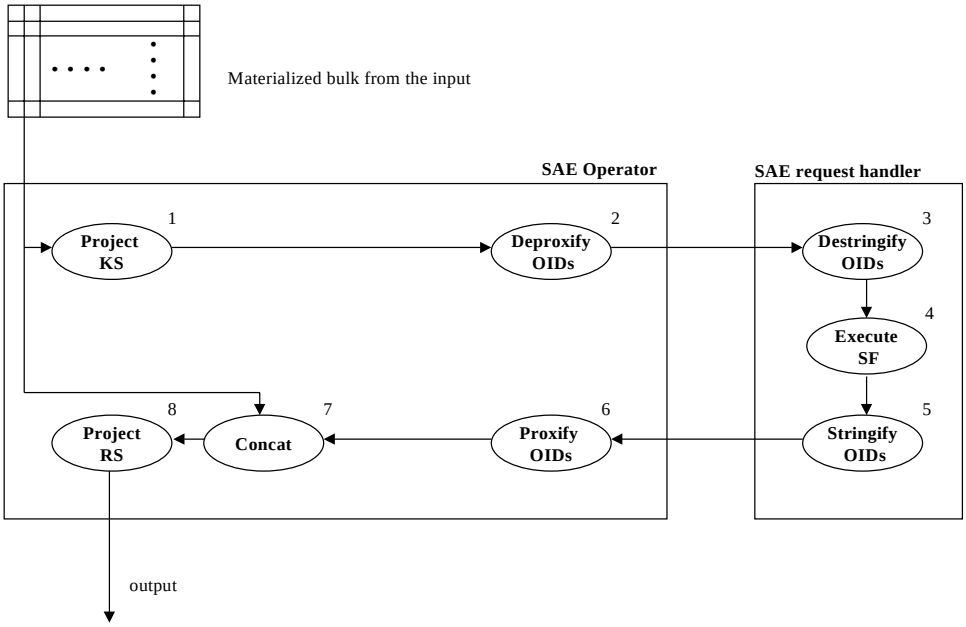


Figure 6.11: Project-concat SAE implementation

SAE operator execution that manages the interaction between the mediator and $DB1$ is represented in Figure 6.10a by the data flows 3 and 4. The input to this execution of the SAE operator is a table containing the values of the variables va and r produced by the execution of the SF $SFdb2$ in the DB2 source. In the example, r ranges over strings, va ranges over integers, and the stored function $fb_{B \to int}$ at $DB1$ has the following values:

$fb_{B \to int}$	
B	**fb(B)**
ib_1	4
ib_2	5
ib_1	6

where ib_k denotes an OID of a B instance. During the execution, the SF $SFdb1$ at DB1 is invoked with an integer as an input. It returns *true* if there exists at least one value in the function $fb_{B \to int}$ that is larger than

the input. The example shows the execution of the SAE operator over 2 bulks of size 4, named in the example as $b1$ and $b2$. The process is shown in parallel for both of the bulks due to space considerations. In reality, the SAE operator is executed sequentially over each of the bulks. Finally, the OID operations have no effect in this type of a setting and are omitted in the example.

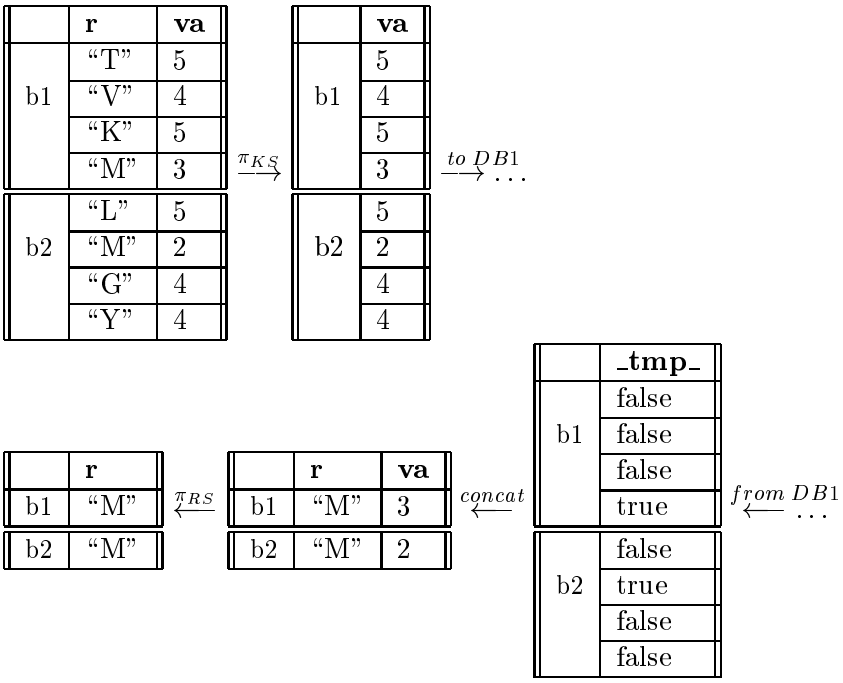


Table 6.1: Example execution of the SAE operator using the project-concat algorithm

In the example, first the projection over the KS variables strips the r values from the input bulks. Next, the bulks are shipped to $DB1$ where the SF $SFdb1$ is executed. The resulting set of boolean values is shipped back to the mediator. The concatenation shown in the example is a special case, when the executed function does not return any data used later in the query processing. In this case, the concatenation of the returned boolean values and the input tuples actually filters the tuples for which the result is *true*. The final projection removes the va values to form the requested result.

The PCA has advantage of improving the naive implementation, while

preserving the simplicity of the processing. All operations are linear in complexity and therefore cheap to perform. Nevertheless, it is inefficient when there is a large percentage of duplicates in the input bulk(s), expensive SF, and/or expensive communication between the servers involved.

The *semi-join* algorithm (SJA) [3], shown in Figure 6.12, improves the performance of the PCA when duplicates are involved. After projecting the input bulk over the columns used as input to the remote SF, SJA performs duplicate removal before shipping the data. When there is a large percentage of duplicates within the bulks, this reduces both the size of the shipped data and the number of executions of the remote SF. The result of the SF execution is shipped back to the calling server where, as in the previous algorithm, the shipped tuples are concatenated to the result of the SF invocation. Next, an equi-join on the *KS* columns is performed over the input bulk and the result of the concatenation. Here, because of the duplicate removal it is not possible to match the tuples by their rank in the bulk.

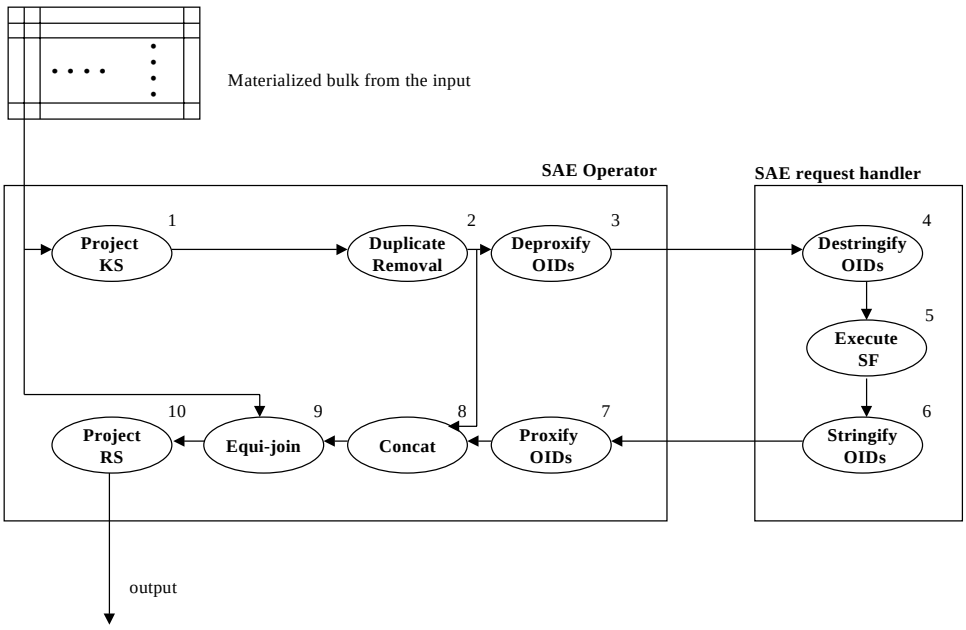


Figure 6.12: SAE by semi-join

The SJA algorithm performs an extra semi-join and a duplicate removal

compared to the PCA algorithm. Both operations can be implemented using hashing in time linear to the size of the input bulks. In the equi-join, it is always cheaper to use the result of the concatenation as an inner set, since it is always smaller or equal to the whole bulk.

The semi-join algorithm benefits from avoiding shipping duplicate entries over the network and executing the SF for them. Nevertheless, this applies to the duplicates only within a single bulk. Duplicates appearing in two different bulks will be shipped and processed separately.

In Table 6.2 an example execution of the SJA is presented in the same scenario as the execution of PCA in Table 6.1. We can note that, due to

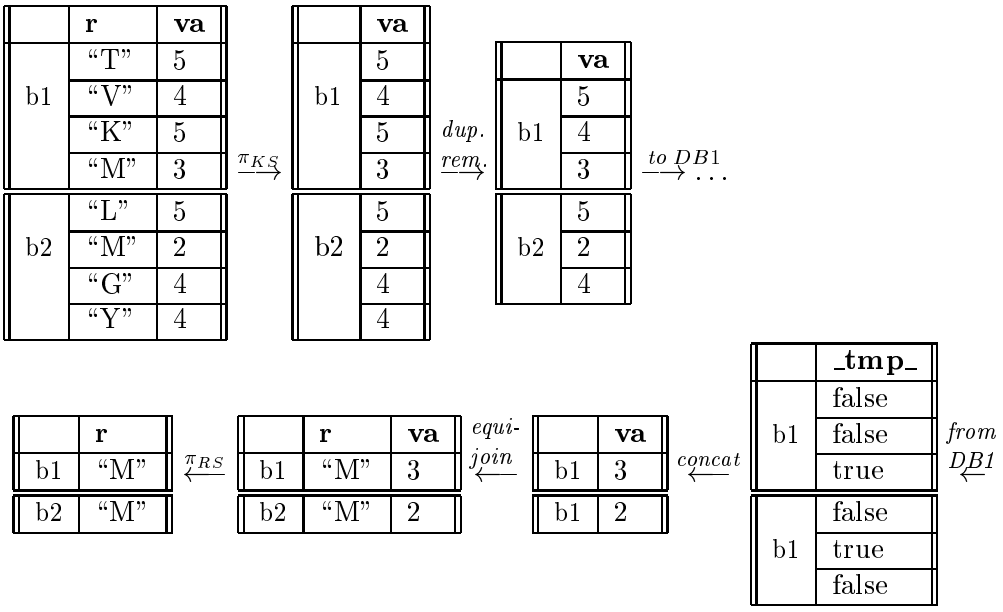


Table 6.2: Example execution of the SAE operator using a semi-join algorithm

the removal of the duplicates within each of the bulks, the size of the table shipped to and from $DB1$ with the SJA example is smaller then with the PCA. The number of the invocations of $SFdb1$ is reduced by the same amount. The added cost is in the two additional phases of duplicate removal and equi-join. Furthermore, these two phases require storage space proportional to the bulk size to store the temporary hash indices built during these phases.

In order to avoid duplicates over different bulks, the algorithm in Figure 6.13, extends the SJA by saving the index built up for the inner relation of the equi-join, between the executions of the SAE operator for different bulks of the input. New entries for the tuples in the input bulk not already in the index are added to the index every time the SAE operator is invoked. The data shipped by the SAE operator is passed through an additional filter where an anti-semi-join is performed over the set already pruned from duplicates. The tuples that are already present in the index are not shipped. If a tuple is in the index, it has already been processed in some of the previous bulks and the result is present in the index. The remaining tuples are shipped to the remote site, where the SF is executed and the result is shipped back as in the previous versions of the algorithm. Next, new entries are added to the index from the returned result. Finally, an equi-join between the input bulk and the index is performed as in the SJA. A comparative execution of this algorithm in the same scenario as the examples of the previous algorithms is presented in Table 6.3. Here, the second bulk is reduced to one tuple before shipping to $DB1$, since the anti-semi-join eliminates the two tuples present in the first bulk.

However, the modification of the SJA requires additional operations that access and update the hash index are introduced. The size of the index in the modified algorithm is proportional to the number of distinct KS tuples. The algorithm can be used even in the case when the whole index is too big to fit in the memory. In this case, when the memory limit is reached, new entries are not added to the index and it serves as a bloom filter [38].

The semi-join with materialized index algorithm (SJMA) in Figure 6.13 does not add substantially to the cost of the SJA, while it offers the possibility for performance improvements. It reduces to the SJA in the case when the whole input is contained in only one bulk. Therefore, we do not consider the SJA algorithm for an SAE operator execution strategy. Only the PCA in Figure 6.11 and the SJMA in Figure 6.13 are considered. An algorithm is selected based on the costs of the "penalties" of each of the algorithms.

The penalty of the PCA lies in the extra shipments and executions of the SF function. If we ignore the OID manipulation operations that are cheap to perform, the extra cost imposed by each duplicate tuple is a sum of the costs to: ship the tuple from the coordinating site (m_c) to the handler site m_h; execute the SF over this tuple; and ship the result back. Assuming that there are PD percentages of duplicates in the input of size IS, and that TS is the tuple of the SF result types, then the penalty of PCA can be expressed

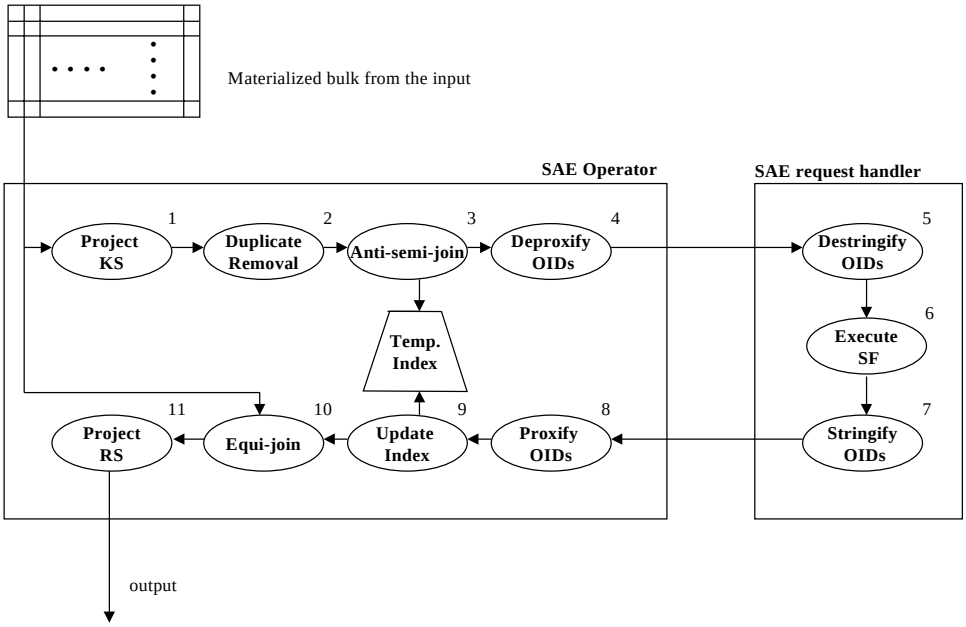


Figure 6.13: SAE by semi-join and a temporary index

as:

$$PD \cdot IS \cdot (W_{mc,mh} \cdot sizeof(KS) + cost(SF) + W_{mc,mh} \cdot sizeof(TS) \cdot selectivity(SF))$$

The SJMA avoids processing of the duplicate tuples at a cost of 4 additional steps: duplicate removal, anti-semi-join, index update and equi-join. Assuming an average index lookup cost of I_l and index update of I_u, an upper limit for the cost of these operations can be expressed as:

$$IS \cdot I_u + IS \cdot I_l + (1 - PD) \cdot IS \cdot I_u + IS \cdot I_l$$

The first term represents the cost of duplicate removal by hashing the input. The second term is an upper limit for the anti-semi-join, if each element is looked up in the materialized index. The actual numbers is lower than the term due to the duplicate removal, but to simplify the calculations we use the size of the whole input. The third term counts the index updates, one for each unique tuple in the projection of the input over the KS column set.

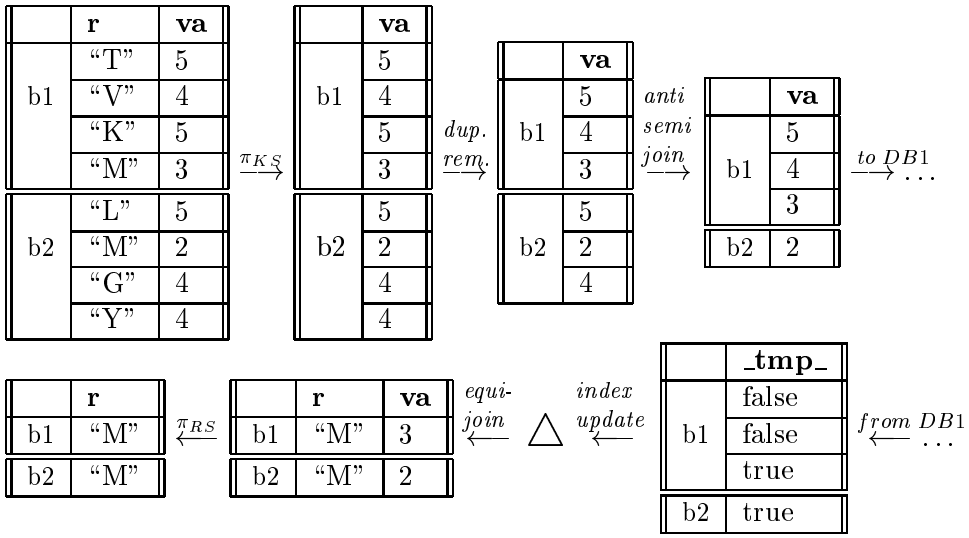


Table 6.3: Example execution of the SAE operator using the semi-join with materialized index algorithm

The cost of the equi-join equals to one index lookup for each input tuple. By comparing the two costs above and simplifying the inequality, we can state the criteria for the use of the SJMA over PCA if the following inequality is satisfied:

$$PD \cdot (W_{mc,mh} \cdot sizeof(KS) + cost(SF) + W_{mc,mh} \cdot sizeof(TS) \cdot selectivity(SF)) > (2 - PD) \cdot I_u + 2 \cdot I_l$$

or

$$PD > 2\frac{(I_u + I_l)}{W_{mc,mh} \cdot (sizeof(KS) + sizeof(TS) \cdot selectivity(SF)) + cost(SF) + I_u}$$

All terms in this equation are already available from the earlier query compilation phases, with the exception of the term PD. This term can be obtained from the estimates of the input local and imported functions. Since some of these come from data sources where good estimates will be impossible to obtain and because of the effects of the query processing steps preceding the one evaluated can increase the errors in the estimates, we propose that the decision is made dynamically at run-time, during the processing of the first bulk of data. For the purpose of determining the PD parameter, the

maximum of 5% or 50 tuples are scanned randomly in the fist input bulk, while the compile-time cost estimates assume the PCA.

Chapter 7

A Survey of Related Approaches

This chapter presents an overview of some research projects with similar aims as the AMOS*II* project. AMOS*II* is related to research in the areas of OO views, data integration, distributed databases and general query processing. We have surveyed the literature on a number of multidatabase integration and OO view projects and compared their approaches to AMOS*II*. To aid the comparison, we first summarize the major features of AMOS*II*:

- A distributed mediator architecture where query plans are generated using a distributed compilation in several communicating mediator and wrapper servers.
- Data integration by reconciled OO views spanning over multiple mediators and specified through declarative OO queries. These views are *capacity augmenting* views, i.e. locally stored attributes can be associated with them.
- Processing and optimization of queries to the reconciled views using OO concepts such as overloading, late binding, and type aware query rewrites.
- Query optimization strategies for efficient processing of queries over a combination of locally stored and reconciled data from external data sources.

7.1 Multidatabase systems

The main purpose of the AMOS*II* project is the development of a system for integration of data in multiple data sources. This section compares the architecture and implementation techniques used in AMOS*II* with other multidatabase integration projects [9, 35, 7, 30, 11, 78, 56, 43, 7, 48, 24].

7.1.1 Disco

The DISCO (Distributed Information SEarch Component) system is based on a centralized mediator-wrapper architecture. Although its primary focus is not on query performance, but on extensibility and partial query evaluation in presence of unavailable data sources, it has many principles in common with the AMOS*II* system.

A special mediator called the *Catalog* keeps information about the available DISCO mediators and wrappers on the network. This service corresponds closely to the name services in AMOS*II*. DISCO is based on the ODMG [9] standard data model and uses OQL and ODL as the query and data definition languages, respectively. One of the central concepts in the ODL is the concept of *type* that has associated an *interface* (structural type description) and an *extent*. In DISCO the concept of a type is extended with these facilities:

- Associating an interface with one or more extents stored in the data sources. The extents contain objects that have a structure as described by the interface.
- Type mappings between types defined in the mediator and the types with extents stored in the data sources, in order to overcome structural differences.

The first extension allows for a type defined in the mediator to draw its extent from a set of data sources. The resulting extent is a union of all the instances in all the sources. The second extension is used to transform the data in the data sources into a common interface.

Data sources are defined by instantiation of the type *Repository*. Repositories are classified into *repository types*. To access a repository of a particular type, a wrapper must be implemented for it. For example, the following expressions define two repositories $r1$ and $r2$ of the same type to be wrapped

with the wrapper $w0$. The data in both repositories has a format described by the interface *Person*. The query retrieves the names of all the persons with salary greater than 10.

```
extent person0 of Person wrapper w0 repository r0;
extent person1 of Person wrapper w0 repository r1;

select p.name
from p in union(person0, person1)
where p.salary > 10;
```

In this example, the extents are named explicitly. Alternatively, they can be specified by meta-data queries that dynamically determine the number of extents to be scanned.

For conflict resolution the user can use the OQL view definition capabilities. Compared to the approach used in AMOS*II* this has the following disadvantages:

1. The reconciliation is performed in the mediator, while AMOS*II* can push the conflict resolution code to the wrappers and the data sources when favorable.

2. The view mechanism (named *sets* in OQL) does not provide OIDs for the view instances and therefore the optimizations based on locally stored data in AMOS*II* are not applicable.

3. The ODL/OQL language does not have conflict resolution constructs as the integration union types (IUTs). This requires the user to manually specify the resolution in case of a data overlap.

The query processing in DISCO is performed over plans described in a formalism called universal abstract machine (UAM) that contains the relational algebra operators extended with primitives for executing parts of the plans in the wrapper. The mediator communicates with the wrapper by using a grammar describing the operator sequences accepted by the wrapper. It can also (in some cases) ask for the cost of a particular operator sequence. This method is more elaborate than the method for the description of data source capabilities in AMOS*II*, but it is more complex and time-consuming, due to the combinatorial nature of the problem of constructing the subplans executed in the wrappers.

Finally, to our knowledge, an implementation of a prototype has been planned, but no experimental results have been reported.

7.1.2 Garlic

The Garlic [35, 36, 64] system, developed at the IBM Almaden Research Center, also has a centralized wrapper-mediator architecture. The system is based on ODMG's OO data model. The data from the wrapped data sources is represented as objects. The OIDs of these objects are constructed from the data source name, the object type, and a set of keys specified for each type retrieved from a data source. Except for the system data and intermediate results, Garlic does not provide facilities for storing local user-defined data, even though it has a fully functional query processor. The primary goals of the Garlic project are:

- To explore how the query optimization techniques based on exhaustive dynamic programming, developed in earlier IBM research prototypes and products, can be used in a data integration scenario.

- To expand these techniques, so that wrappers for different data source types can be easily specified, modified, and added to the system.

At the heart of the Garlic system is the *query service* facility. This facility is divided into two units: (i) a query language processor, and (ii) a distributed query execution engine. The query language processor performs tasks that correspond to some of the the calculus-related phases in the query processor of AMOS*II* (semantic checking, rewrite, etc.). The second unit performs cost-based optimization and outputs an executable query execution plan.

The query optimizer in Garlic is based on dynamic programming. The optimizer builds plans of gradually increasing sizes, by adding POPs (Plan OPerators) to already built partial plans. The POPs can be relational algebra operators, operators for storing and retrieving data in temporary tables, and operators for accessing the wrappers. During the search, the optimizer prunes the plans that are more expensive than other plans representing the same subquery. In addition to the composition of the plans, Garlic takes into account the location of the result of the plan execution. Two plans computing the same subquery, but placing the result in two different sites are not pruned from the system.

POPs are added to already constructed partial plans using STARS - (STrategy Alternative Rules). Each rule describes how a new plan is constructed from one or more partial plans. Each rule has a condition attached that guards its triggering. The rules create POPs that are executed locally,

or the *PushDown* POP that executes a subquery in a wrapper. For the *select-project-join* queries there are three main STAR types (named *STAR roots*) for: *access* (scan), *join*, and *finish* (plan completions as e.g. projections). The two first STARs can produce plans that execute either in the mediator or in the data sources, while the third one is always executed in Garlic. An illustrative example of a STAR root is the *join* root that is translated into three different POPs: *ReproJoin* - executed in a wrapper, *NestedLoopJoin* - executed in Garlic over materialized operands, and *BindJoin* - a semi-join-like POP where the outer table is sent one tuple at a time to the site of the inner table, retrieving the matching tuples.

The functionality of Garlic roughly corresponds to the multidatabase query engine in AMOS*II*. An important difference is that Garlic does not store local data. The generated OIDs are used only to access the data in the data sources. This makes the techniques for optimization of queries over a combination of locally stored and imported data in AMOS*II* inapplicable. It seems that the Garlic OIDs are used only internally in the query processor, and possibly as object handles for user requests over individual objects. Furthermore, Garlic has no facilities equivalent to the IUTs in AMOS*II*.

Another difference between the two approaches is that Garlic is based on a centralized query compilation and execution architecture, while AMOS*II* is based on a query processor that performs distributed query compilation in the network of wrappers and other mediators. Therefore, we cannot directly compare the POP formalism of Garlic with the decomposition tree (DcT) formalism of AMOS*II*, designed to distribute queries over multiple AMOS*II* servers. By contrast, a Garlic system treats another Garlic system as a (relational) data source. Therefore strategies as achieved by the DcT distribution are not explored. Although, it can be supposed that STAR rules can be formed to take advantage of the intermediate result materialization capabilities of the Garlic mediators in order to employ similar strategies as in AMOS*II* in a network of Garlic mediators, this approach has not been pursued in the reported work.

One limitation of the current implementation of AMOS*II* is that it always pushes the joins to a data source where all its operands are available. Garlic, on the other hand, also considers plans that perform the join in the mediator. Our on-going work includes expansion of AMOS*II* to consider such query execution plans.

7.1.3 Pegasus

The goal of the Pegasus project is to develop a heterogeneous information and process flow management system (HP-MS). This project was started in the early 1990s at the HP Labs in Palo Alto.

Pegasus is a fully fledged database management system. The focus of the project is on integration of relational databases, multimedia databases, and legacy applications. The three main goals of the Pegasus project are:

- seamless integration of external schemas with the local database
- efficient query processing
- workflow management

Pegasus originates in the same data model as AMOS*II*: the Iris OO data model [21], an OO extension of DAPLEX [71]. Earlier versions of Pegasus used the HOSQL language that is an extension of the language OSQL [53] used in the Iris system. OSQL has also served as a basis for the AMOS*II* query language AMOSQL. More recently Pegasus has been shifted to an SQL3 based language SQL3+. This language extends the SQL3 standard with data integration facilities.

Although the terminology differs greatly, the architecture of the Pegasus system is similar to the mediator-wrapper architecture used in AMOS*II*, but it is not distributed. The core of Pegasus corresponds to the mediator services. External data sources are named External Data Resource Management Systems (EDRMSs). The interaction with the EDRMSs is performed using a module *Pegasus Agent* (PA) that also has some processing capabilities. The PA process is intended to run on the same machine as the EDRMS it serves. The functionality of the PA is similar to a functionality of the wrappers in the AMOS*II* architecture. Nevertheless, a PA is not a fully-fledged Pegasus server in the centralized architecture of Pegasus. This is one of the most important differences with the AMOS*II* architecture.

The data integration facilities of Pegasus are named using the distributed database terminology. *Foreign tables* are imported from declared data sources. In the following example, summarized from [7], first a DB2 relational data source named $DB1$ is defined and then a table is imported and bound to the type $Programmer$ in Pegasus.

```
REGISTER RELATIONAL DB2
```

```
        DATASOURCE db1 AT 'smith@host1' AS Pdb;

CREATE TYPE Programmer WITH OID VISIBLE
  ( Prog_id INTEGER,
    Ssn     INTEGER,
    Name    CHAR,
    Salary  INTEGER);

CREATE TABLE ProgrammerTable (Dcn: Programmer) AS IMPORTED
   FROM RELATIONAL DATASOURCE Pdb RELATION Programmer
   WITH OID PRODUCING BY (Prog_id)
   (Prog_id AS MATCHING  Prog_id,
    Ssn     AS MATCHING  Ssn,
    Salary  AS MATCHING  Salary,
    Name    AS MATCHING  Name);
```

The resulting table has a one-to-one correspondence with the table in the relational database. The OIDs of the new type are formed using the *Prog_id* column from the relational database.

Imported tables can be integrated with locally defined tables, as well as tables imported from other data sources, by two mechanisms:

- integrated views
- adding columns of one table to another

The first mechanism allows for merging horizontally fragmented tables, while the second is used for merging vertically fragmented tables. As opposed to the classical distributed database work, here the fragments are maintained by autonomous database systems. The following example from [7] defines first a supertype over the types *Programmer* and *Engineer*, and then an integrated view over the tables corresponding to this types:

```
CREATE TYPE Employee
   OVER Programmer WITH Prog_id AS Emp_id,
        Engineer   WITH Eng_id  AS Emp_id,
   (Emp_id INTEGER,
    Ssn     INTEGER,
    Name    CHAR,
    Salary  INTEGER);
```

```
CREATE VIEW EmployeeTable (Dcn Employee)
   AS SELECT * FROM ProgrammerTable UNION ALL
      SELECT * FROM EngineerTable;

DEFINE ROW EQIVALENCE FOR EmployeeTable ON
   (Tp ProgrammerTable, Te EngineerTable)
     BY Tp.Dcn->Ssn = Te.Dcn->Ssn;

DEFINE RECONCILER ON EmployeeTable.Dcn->Ssn
   (ProgrammerTable, EngineerTable)
      RETURNS INTEGER USING DISAMB_SUM
```

The ROW EQUIVALENCE clause defines the equality condition for the rows of the defined view. Rows that satisfy this condition will be treated as one in the resulting tables. Possible conflicts in the values of the other columns are resolved by RECONCILER definitions. These can be system specified as, for example DISAMB_SUM returning the sum of the input values, or user defined derived functions. Although the view definition uses the UNION operator, the ROW EQUIVALENCE clause enforces outer-join semantics and processing.

The processing of the queries over the integrated view proceeds in three phases. The first phase performs query rewrites to transform the query from using the integrated view to the imported tables. Then, the query processor identifies portions of the query tree that can be evaluated in a single EDRMS and converts them into *Virtual Tables* (VT) that encapsulate the operations performed in the EDRMS. The resulting query tree has VTs or locally stored tables as leaves and internal nodes representing operators of extended relational algebra. The extensions deal, among other things, with object-oriented concepts, constraints and reconciliation. Some of the query rewrite rules applied in this phase are: [7]:

- Push as many as possible of the operations into the EDRMS.
- Push up the reconcile operations in order to place the join operations close to the outer-joins.
- Combine joins with the outer-joins in order to make the inputs to the outer-join smaller.

- Transform the outer-joins to left- or right-outer-joins, or to ordinary joins when some of the other query predicates use some attributes not present is the both of the joined tables. Since the language is null-intolerant (a predicate evaluates to false when a part of it is null), this eliminates the parts of the outer-join where this predicate is not present.

The second query processing phase builds a left-deep query tree using a cost-based method. The costs of the VTs are obtained using elaborate cost model for the operations performed in the EDRMS and calibration of the data sources [15].

The left-deep query tree generated in the first two phases is rebalanced in the third phase. The rebalancing operations are performed at certain points of the tree and are based on the associativity and commutativity properties of the join and cross-product operators.

Each of the three query processing phases in Pegasus can be related to a phase in the query processing in AMOS*II*. The first phase corresponds to the calculus generation and rewrite, with a difference that the rewrites in AMOS*II* reduce the number of predicates, while in Pegasus they perform reordering of the operators that influences their order of execution in the final execution plan. Techniques, as in AMOS*II*, that take advantage of the types of the query variables to reduce the query size are not described.

In processing of queries over the integrated views, Pegasus keeps the outer-joins as a single operation, and later in the query it performs a correction of the result by reconciliation operators. This approach has the advantage of keeping the queries compact, but it does not take advantage of the selections stated over reconciled functions. We believe that these kinds of selections appear often in queries over the integration views.

In AMOS*II*, on the other hand, the outer-join and the reconciliation is broken into up to three cases: one join and two anti-semi-joins, each processed separately. This allows selections specified over the reconciled functions to be pushed all the way down to the data sources in the two anti-semi-joins cases. In the join case, the optimizer might be able to push the selections down to the data sources when the reconciliation is defined using function values from only one of the data sources. Even when this is not the case, the join still generates smaller intermediate results than the full outer-join, in particular when the overlap is small. The size of the result and the data shipped to perform the join has a maximum size proportional to the size of the smaller

of the integrated extents. The outer join produces an intermediate result that is of size equal to the sum of the sizes of the integrated extents.

Another disadvantage of performing the reconciliation late in the query execution is that the reconciliation operator requires its whole input, in this case an outer join of the integrated tables, to be materialized before the processing starts. This prevents streamed execution and might pose problems in cases when the intermediate results are too big to fit into the integration system memory.

In [16] the problem of parametrized queries to the data sources is explored in the context of a study of different join strategies in multidatabase systems. A hash join a strategy is proposed that is similar to the index materialization used in AMOS*II*. However, this research concentrates on individual join operators and ignores the query context that might contain useful predicates to reduce the hash index size. The authors note that this is a hard problem, and instead use the maximum and the minimum values of the local table join column (the input bulk in AMOS*II*) to perform range selection of the joined values, in order to reduce the materialized hash index. The strategy proposed in AMOS*II* successfully solves the problem of utilization of the useful query predicates in the index materialization. The method for reducing the index size used in Pegasus can easily be added to AMOS*II*.

Due to its centralized architecture the rebalanced trees in Pegasus are constructed and stored in a single system. Distributed architecture is one of the future topics of the Pegasus project [7]

7.1.4 TSIMMIS

The TSIMMIS system - The Stanford-IBM Manager of Multiple Information Sources [30] is a continuation of the Lightweight Object Repository (LORE) project, and is aimed for integration of a large number of structured and unstructured data sources. The basis for the integration is a common data model named *object exchange model* (OEM). The idea behind the OEM is to provide as simple as possible, but complete facilities for data integration. Although OEM is not a fully-fledged OO model, the basic entity in OEM is the "object". Each object is composed of four elements: *label*, *type*, *value*, and $object-id$. As opposed to other OO models, the OEM is *self-descriptive*. The type and the label of an object contains the information usually stored in a database schema. Actually, the notion of schema is absent in the OEM. The

authors claim that the labels can be used not only for naming the objects, but also for inferring semantics that can be used in the data integration process. The value field of an object can contain a collection of literals or nested objects, thus creating a graph-like database structure.

To query a database described in OEM, a client can issue a query in a query language named OEM-QL. This query language adopts the OQL (and SQL) syntax style and is based on the **select-from-where** clause. The semantics, however, is based on the OEM model. The path expressions in OEM-QL allow queries over the labeled graph that contain wildcards and other regular expressions that make the navigation easier. As a result, an OEM-QL query returns an OEM graph.

The TSIMMIS project uses a centralized mediator/wrapper data integration architecture. Mediators can fetch and combine data from wrapped data sources. However, unlike AMOS*II*, the TSIMMIS wrappers do not have a complete query processor and data store. The emphasis in TSIMMIS has been to enable easy wrapper and mediator generation, using a *mediator specification language* (MSL), rather than on query performance as in our work. MSL is a rule based language where the input query is matched against a rule specification. In the wrapper definition, when a match is found, data source specific code specified within the rule is executed in order to retrieve the relevant data from the data sources. The data source capabilities mechanism in AMOS*II* are more elaborate and perform cost-based and heuristic optimizations that are not applied in TSIMMIS. Also, the OO transformations used in AMOS*II* are, due to the differences in the CDM, are inapplicable in TSIMMIS. The mediator generation system in TSIMMIS allows for joins, but does not consider integration operators for resolving conflicts in overlapping data as in AMOS*II*. Furthermore, to our knowledge, the TSIMMIS project has not reported performance evaluation of the execution of queries over views defined over data combined from the mediator and different data sources.

7.1.5 Multibase

The Multibase project [11, 12, 13, 14] is a pioneering work on integration of data in multiple databases. As in AMOS*II*, the Multibase system is based on a derivative of the DAPLEX data model [71] extended with generalization. Data integration is performed by defining *generalized types* as supertypes of existing database types. For the generalized types, derived functions based

on the functions of the subtypes can be defined to reconcile the data in the integrated databases. These features are closely related to the *functions* clause in the IUT definitions in AMOS*II*.

Query transformations are used to transform a query over the generalized types into a set of queries over the local schemas. These query transformations break down the outer-joins and the reconciliation functions into queries the over disjoint parts of the integrated relations, using joins and anti-semi-joins. The approach allows for similar optimization techniques as the ones used in AMOS*II* for optimizing queries over IUTs having no locally stored functions. Nevertheless, the method used in Multibase is not based on system predefined types and the properties of type hierarchies, making the query analysis and optimization more complicated. Furthermore, the project does not explore combining optimization by generalization with constructs such as the DTs in AMOS*II*.

Another important difference between the two systems is that the AMOS*II* data model is OO, while the Multibase system lacks OIDs. The lack of OIDs disallows both materialization of the instances of the integrated types and seamless mixing of local data with data retrieved from various data sources. Locally stored data in not considered in this project.

In [12] it is also identified that for selections over the reconciled functions, the two anti-semi-joins usually will be able to take advantage of these. The authors describe three optimization techniques to push the selections through the most common aggregations used in reconciliation of function values of overlapping data. Nevertheless, they note that these techniques apply to very limited number of cases. Therefore, we have chosen not to pursue this approach in AMOS*II*.

Finally, to the extent of the reported work available to us, the benefits of the proposed optimization techniques have not been quantified by experimental results.

7.1.6 Data Joiner

The IBM DataJoiner [82, 78] is a state-of-the-art commercial product targeted for integration of relational databases of different vendors. As opposed to the previous generation integration tools that provide only a gateway for retrieving data stored in multiple vendor databases, the DataJoiner has a full-scale distributed query processor capable of pushing down whole subqueries in to the connected databases. DataJoiner also is a fully-fledged DB2

database.

DataJoiner's query optimizer has a detailed knowledge of the strategies used in the database engines supported as data sources. This meta-data, stored in a *Server Attribute Tables* (SATs) includes information such as the vendors' join implementations, index usage, type of query trees used in the optimization, specifics of the SQL dialect, etc.

As a complement to the SAT table information, the system uses sampling techniques or catalog queries to build locally stored statistics about the characteristics of the tables imported from the data sources. Using this information, the DataJoiner optimizer is capable of precise estimates of the costs of the subqueries pushed to the source databases.

The DataJoiner query optimizer is an extension of the DB2/CS Starburst optimizer. It enumerates all the possible plans using a dynamic programming approach, as in AMOS*II*. The suboptimal plans are pruned. The generated plans explore both performing the operations in the integrator, or if possible, in the data sources.

The portions of the plans pushed to the data sources are translated to SQL that closely resembles the execution strategy used by the local query processor. By this, the DataJoiner takes over the optimization decisions from the relational database used as data sources. The authors of the system claim that in many cases they generate queries that perform better than if the original query was executed directly in the system. An industry report [65] comparing the DataJoiner with two other products in the same area, sets the performance and the functionality of this product high above the other two products.

As opposed to AMOS*II*, DataJoiner is a purely relational product, it does not provide reconciliation facilities, and it has a centralized architecture.

7.1.7 MIND

The MIND (Middle-East Turkish University Interoperable DBMS) prototype [56, 60, 19] is based on the OMG distributed object management architecture. The system is implemented around DEC's ObjectBroker ORB. Various relational databases from different vendors are connected to the system using an interface defined in IDL. Two interfaces play a major role in the MIND integration architecture: the Global Database Agent (GDA) and the Local Database Agent (LDA). For each session with a client, the GDA is instantiated in a server CORBA object that handles the requests for

the client. The CORBA architecture provides location transparency for the GDA objects (GDAO). A GDAO contains a Global Transaction Manager Object and a Global Query Processor Object (GQPO). The latter performs the query decomposition and sends an executable plan for execution to the former. The LDA objects (LDAOs) manage the submissions of the operations to the relational data sources and transaction management. The tasks of the GDAOs and LDAOs can be related to the tasks of the mediator and wrapper in the mediator-wrapper architecture.

The schema integration process is based on a typical four schema transformation layers: local, export, global and external schema in order from the individual data sources to the applications. The focus of these transformations is on resolution of the class structural conflicts and class extent conflicts, while preserving the autonomy of the sources.

The conflict resolution is specified by a *mapping* definition. The following example [56] illustrates an integration of departments tables from three databases (*dept@DB1, division@DB2 and department@DB3*):

```
/* global schema: department(dept_no, dept_name, address) */
/* local schemas: DB1: dept(dno, dname)
  DB2: division(dno, dname, location)
  DB3: department(dno, deptname, address) */


mapping department {
  origin
     DB1: dept d1,
     DB2: division d2,
     DB3: department d3;
  def_ext dept_ext  as
     select * from d1, d2, d3 where d1.dno *= d2.dno
                                and d2.dno*=d3.dno;
  def_att dept_no   as
     select d1.dno, d2.dno, d3.dno from d1, d2, d3;
  def_att dept_name as
     select d1.dname, d2.dname, d3.deptname from d1, d2, d3;
  def_att address   as
     select d2.location, d3.address from d2, d3; }
```

The *mapping* clause specifies the data sources, the extent of the new class, and the correspondence of the local tables' attributes to the attributes of the global table. Note that the *= operator denotes an outer-join.

The goal of the query decomposition is to produce:

- A set of single data source queries that retrieve the needed data from each of the involved data sources
- A set of post-processing operations executed in the GDAO that produce the query result from the intermediate results sent by the data sources.

The set of single data source subqueries is produced by instantiating the global schema query for each of the data sources. One limitation of this process as described in [56] is that for a join over two integrated tables, the generated single site queries assume that the joins are performed only over fragments (integrated tables) located at a same data source. For example, a query where the *dept* type defined above is joined with an *emp* table integrating data from the same three sources will produce subqueries that explore only entries where the local employee tables join with the local department tables. Cross-source strategies (e.g. where an employee at DB1 works at a department stored at DB2) are not considered. Although this conforms with the probable intended semantics of the example above, in general these kinds of simplifications are application-dependent and, in our opinion, should be inferred on the basis of declared database constraints.

In the post-processing phase, the GDAO performs operations such as outer-joins and joins to build the final result from the intermediate results returned by the data sources. The execution plan for this phase is generated by using a dynamic and heuristics-based query optimization approach that takes into account the actual load of the systems during the query execution time.

Like the other systems that perform the reconciliation in the final phases of the query processing, the method used in MIND suffers from not being able to use the selections based on reconciled functions early in the query processing. Also, although the requests to the data sources can be executed in parallel, the reconciliation process in the mediator has to wait until all the inputs are materialized, before emitting the first result tuple.

Another difference between MIND and AMOS*II* is that MIND's integration facilities do not provide means for materializing OIDs for the data

from the data sources and augmenting the views over this data with locally defined attributes.

7.1.8 IRO-DB

The IRO-DB project (Interoperable Relational and Object-Oriented Databases - ESPRIT - III P8629) [20, 73, 25] developed tools for unified access to a number of relational and OO databases. The system is based on the ODMG standard data model and the query language OQL. The architecture of IRO-DB is divided into three layers:

- The **Local layer** represents the data sources wrapped by Local Database Adapters (LDA) that provide ODMG/ODL mapping to the schema and OQL access to the data in the sources. This layer also generates OIDs for the instances in the OO CDM that correspond to the instances in the data sources.

- The **Communication Layer** performs the transfer of objects and OQL queries between the server and the client sites. The protocol used is an OO extension of the remote database access (RDA) standard, OORDA. The main purpose of the communication layer is to allow the interoperable layer to communicate with the local layer, but it can also be used by the applications to directly access the data sources via an OO extension of SQL CLI.

- The **Interoperable Layer** provides the application with means of integrated access to multiple remote databases. Its functionality can be divided into two parts: an interoperable DBMS (IRO-DBMS) that supports the use and maintenance of an interoperable (global) schema, and tools for aiding the building of an interoperable schema (Integrators Workbench).

Compared with the wrapper-mediator architecture, the interoperable layer provides services that correspond to the mediator services, while the local layer corresponds to the wrapper. In the following, a description of the IRO-DBMS is presented. The IRO-DBMS also consists of several functional units:

- The *API generator* generates an ODMG compliant C++ API from the integrated schema to be used by applications that access this schema.

- The *global transaction manager* implements the nested transaction protocol of the ODMG standard.
- The *global parser and processor* takes a text representation of an OQL query and returns the result of its execution over the interoperable schema. The features of this unit in relation to the AMOS*II* system will be explored in greater detail in the rest of this section.
- The *global data repository* stores and provides the rest of the system with an export schema description, a description of the interoperable schema, schema localization information, and a description of the mappings between the export and the interoperable schemas.

The data integration schema in IRO-DB is specified by three layers of class mappings. Each class to be exported by a data source is named an *external class*. In the interoperable system each external class of interest has a corresponding *imported class* serving a similar purpose as the proxy types in AMOS*II*. The actual integration is performed by defining *derived classes*. The interoperable system can also host locally stored data organized into *standard classes*. The following example illustrates the use of the *mapping* construct used for defining derived classes. In the example, first two imported classes, *S1_PART* representing the table *part* at the source *S1*, and *S2_PART* representing the table *prt* at the source *S2*, are defined. The mapping clause defines the extent of the derived class *PART* and its attributes using query expressions [73]:

```
mapping imported S1_PART{
    origin S1::PART   orig;}

mapping imported S2_PART{
    origin S2::PRT    orig;}

mapping PART {
  origin S1_PART sorig;
  origin  S2_PART iorig;
  def_extent parts as select PART(sorig: s_i, iorig: i_i)
                        from s_i in s1_parts, i_i in s2_ptrs
                        where s_i.part_id = i_i.prt_id;
  def_att part_id  as this.sorig.part_id;
  def_att upd_date as this.sorig.upd_date;
```

```
def_att description as this.iorig.ptr_tpflg;}
```

Since the derived classes can use a general query to draw their extents from the *origin* classes, they can be used for functionality that corresponds to the DTs in AMOS*II*. Extent definitions with outer-join conditions could be used to define constructs similar to the IUTs, but this are not elaborated in the IRO-DB reports available, nor are special query processing techniques to support this type of operators presented. Also, the derived classes are not placed in the class/type hierarchy as are the DTs and IUTs in AMOS*II*.

As AMOS*II*, IRO-DB also uses proxy objects in the interoperable system to represent objects in the data sources. The same mechanism is used for the derived classes. This mechanism is similar to the coercion mechanism used for the AMOS*II* DTs. However, the AMOS*II* IUTs are different. When IUTs are used in AMOS*II*, no new OIDs are created (and no coercion is used) since the extent of the IUT is a union of disjunctive sets of object instances of the auxiliary subtypes. Another difference in the proxy manipulation is that in AMOS*II* the proxy OIDs are generated in the mediator corresponding to the interoperable layer in IRO-DB, while in IRO-DB these are generated by the LDAs. This leads different internal representation of the OIDs of the standard class objects and the OIDs of the imported class objects. The objects of the later type have longer OIDs storing redundant class and source information that in AMOS*II* is stored in the interoperable schema as a property of the imported classes.

The handling of the requests for object attribute values also differs considerably between the systems. In IRO-DB when a proxy object is used, the systems accesses the data source and materializes in the interoperable database (also known as home database) all the attributes of the object. Possible references to other global objects are replaced by global OIDs, if these objects are already in the home database. Otherwise, these objects are retrieved first and then assigned global OIDs. The process proceeds until no unresolved object references exist in the materialized object graph. After this materialization, the queries using this object within a single transaction access the local copy. The home database thus acts as an object cache of all integrated data in IRO-DB.

IRO-DB queries can be processed using two modes of operation: (i) ad-hoc queries can be processed by ignoring the current contents of the home-database and rematerializing there a superset of the object instances needed for the query evaluation before processing the query over the cache; (ii) long-

transaction queries that are more likely to access the same objects more than once, and therefore the query processor tries to materialize only the objects missing in the home-database with the cost of more complicated processing.

Compared to this mechanism, AMOS*II* uses selective retrieval of the proxy object function values that are used in the queries. This approach does not pay the penalty of retrieving some (possibly large) unused attributes and long chains of object referenced from the first object. In conjunctive AMOS*II* queries, the calculus rewrites remove the common subexpressions that produce most of the repeated accesses to a single function. It is possible, in rare cases, that the same function values are retrieved twice within same conjunctive query that has two variables ranging over a single proxy type. This is rare and the penalty is big only when the function values are very large or the function invocation is very costly. Prefetching of proxy function values can be more useful in AMOS*II* in the context of disjunctive queries as the one used when processing of queries over the IUTs. However, the analysis of these queries is much more complex than the analysis of conjunctive queries. Such features are one of the future research topics in the AMOS*II* project.

Some issues that are addressed in AMOS*II*, but to our knowledge, are not considered in IRO-DB are: (i) optimization of queries over combined local and imported data, (ii) queries with outer-joins and complex reconciliation functions, (iii) queries over hierarchies of derived classes and (iv) experimental study of the performance of the presented query processing strategies. The IRO-DB project is succeeded by the MIRO-Web project [25].

7.1.9 DIOM

The *Distributed Interoperable Model* (DIOM) project [50, 63] has developed a distributed mediation framework based on the ODMG-93 data model. The goal of this project is to provide a scalable platform for uniform access to autonomous and heterogeneous systems based on evolving and composable mediators. A network of domain-specific mediators is deployed to support application access to the data in the data sources. Each mediator is instantiated from a meta-mediator by defining an integration schema. The meta-mediator architecture, *Diorama*, consists of two layers: a mediator layer and a wrapper layer. The mediator layer contains:

- *Interface manager:* provides a GUI interface and an API that expose the mediator functionality to the users.

- *Distributed query mediation services:* provides source selection, query decomposition, parallel access plan generation and result assembly.
- *Runtime supervisor:* executes subqueries in the wrappers.
- *Information source catalog manager:* manages the data source information and interface repository meta-data. Communicates with the local implementation repository in the wrapper layer in the management of the local wrappers data.

The wrapper layer has the following components:

- *Query wrapper service manager:* receives the requests from the runtime supervisor, translates the query in DIOM to a query in a local language using the data in the implementation repository, executes the subquery and returns results.
- *Implementation repository manager:* maintains the correspondence between the source data and its DIOM representation.

The unified view of the data in the repositories is built using *meta-operations* applied to *base interfaces* representing data in the data sources and *compound interfaces* built recursively by meta-operations. There are four meta operations in DIOM:

- *Aggregation* allows composition of a new interface based on a number of existing interfaces. The new interface can reference the existing interfaces when defining attributes. For example, a new interface *employment* can be defined that links employees from one database with departments from another.
- *Generalization* is used to merger several semantically similar interfaces into one. The new interface abstracts some common properties/attributes of the merged interfaces. An instance union semantics is used that does not provide for overlap resolution.
- *Specialization* creates a new interface by adding new attributes or operations to an existing interface.
- *Import/Hide* is used to import portions of schema from other DIOM mediators. It preserves the closure of the imported subschema by implicitly importing the types of the attributes and operations of the

explicitly imported types. The *hide* clause can be used to exclude certain attributes from importing. The imported interfaces can also state their relationship in the exporting interface hierarchy using the *ISA* keyword. This meta-operation corresponds to the proxy type mechanism in AMOS*II*.

Queries over integrated schemas are posed in a language named *interface query language* (IQL). The syntax of IQL is similar to the one proposed by the ODMG-93 OQL. One distinction is the *target* clause that is added to the *select-from-where* block to describe the possible data sources where the query is applied. The authors also propose a mechanism for automatic detection of equi-joins among the object types used in the *from* clause, to relieve the user of specifying obvious conditions in the *where* clause.

The IQL queries are processed in 5 phases:

- *Query routing* This phase selects the relevant information sources from the set of all available sources, by mapping the domain model terminology to the source model terminology.
- *Query Decomposition* Partitions a query expressed over a compound interface into queries over the basic interfaces used in the definition of the compound interface. Interfaces defined using aggregation and generalization meta-operations are substituted by n-ary join and union expressions respectively. Selections and projections are pushed down to the sources while joins that are performed at the same site are grouped together.
- *Parallel access plan generation* The query scheduling strategy described in [63] first builds a join operator query tree (schedule) using a heuristics approach, and then assigns execution sites to the join operators using an exhaustive cost-based search. AMOS*II*, on the other hand, performs a cost-based schedule composition and heuristic execution site assignment. Furthermore, the scheduling process in DIOM is centrally performed, and no distinction is made between the data sources and the mediators in the optimization framework, ignoring thus the problem of having sources with different capabilities. DIOM uses a parallel execution cost model. This is one of the current research issues in the AMOS*II* project.
- *Subquery Translation and Execution* performs tasks similar to that of the wrapper layer in AMOS*II*.

- *Query result packaging and assembly* This phase uses the results of the subqueries generated by the query decomposition to assemble the result required by the user.

DIOM does not specify constructs for resolving conflicts in an overlap among the data in the data sources. Also, no strategies to optimize queries over a combination of local and reconciled data are presented. Finally, no quantification of the benefits of the proposed strategies is presented in the available DIOM project reports.

7.1.10 UNISQL

The UNISQL [43] system is one of the first commercial products that provide views for database integration. The data integration views are built of *virtual classes* that correspond to the AMOS*II* DTs, but are organized in a separate hierarchy. The virtual class instances inherit the OIDs from the ordinary class objects. This does not provide for definition of stored functions over virtual classes defined by multiple inheritance, as in AMOS*II*. In UNISQL there is no mechanism corresponding to the IUTs in AMOS*II*, but rather a set of queries can be used to specify a virtual class as a union of other classes. This relationship is not included in the type hierarchy, imposing two different kinds of dependencies among the virtual classes.

7.1.11 Remote-Exchange

The remote exchange project at University of Southern California [24] uses a CDM similar to the one used AMOS*II* to establish a framework for instance and behavior sharing. Three dimensions of freedom are explored for function application in a federated database environment: the location of the function (local or remote), the location of the arguments (local or remote), and the type of the function (stored or computed). Each case is elaborated and an abstract implementation description is given. Most of the cases correspond to the ones found in AMOS*II*, although the terminology differs considerably. One case not covered in AMOS*II* is the execution of remote derived functions over local objects. In this case the execution is performed at a remote site, where each function call in the definition of the derived function is trapped and a callback is issued to the local system for the needed argument values. This requires that the function calls used in the remote derived function evaluated over a local object have exactly the same name and semantics in

the remote database as they have in the local one, limiting the use of the feature.

The disadvantages of the Remote-Exchange approach is that it forces late binding on every function execution that might need to be done remotely. Next, all remote operations are performed on an instance basis by performing an RPC for each individual instance. Another performance degradation is caused by the size of the surrogate identifiers for remote instances that are 300 bytes long and contain all the information needed to perform the remote function evaluation over the instance. Data integration features such as the DTs and IUTs in AMOS*II* are not described.

7.1.12 Myriad

Myriad [48, 49] is a federated database project developed at the University of Minnesota. The federation is defined as an integrated database with a global schema consisting of a set of global relations. This relational schema can be specified over data tables stored locally, as well as in other relational database systems accessed by gateways. An SQL-like language is used to query the integrated database schema. The goal of this project is to provide global query processing and transaction management over a set of autonomous and heterogeneous relational DBMS storing pre-existing data.

The global schema is generated from the export schemas by a specification based on outer-joins and the *generalized attribute derivation* GAD operator. The GAD operator is a reconciliation specification mechanism by which the local database attributes are mapped to a corresponding global schema attribute. The following example, based on an example in [48], defines a global *res* relation based on a three relations *res_A, res_B* and *res_C* stored in the relational databases *A, B* and *C* accordingly:

```
RES <-- GAD(
    OUTERJOIN({res_A, res_B, res_C},
       (res_A.rname = res_B.rname)
       (res_B.rname = res_C.rname)
       (res_C.rname = res_A.rname)),
  (rname   F_key(res_A.rname, res_B.rname, res_C.rname))
  (rating  F_avg(res_A.rating, res_B.rating))
  (cost    F_max(res_A.cost, res_B.cost)))
```

In the example, it is assumed that the three data sources have equal schemas. The outer-join is performed over the attribute *rname*. The reconciliation is

performed by the system-defined functions F_fn. They perform the usual aggregation operations, except the F_key function that picks the first of the arguments with a non-null value.

Global queries are accepted by a *Federated Query Manager* (FQM), that performs the translation into executable plans executed by *Federated Transaction Management*. On the data source side, the function of the wrapper is performed by a *Federated Transaction Agent* accepting the requests and invokes the *Federated Query Agent* that processes the requests and handles the communication with the data source.

The FQM translates a query over the global schema into a set of queries over individual export schemas and a set of result assembling operations executed in the FQM. Although a fully-fledged query optimization module is not implemented [48], [49] present an extensive study of optimization of queries including several outer-joins and GAD operators. This work manipulates the queries in a formalism named *constrained query trees* (CQT). CQTs are relational operator query trees extended with n-ary union, join and outer-join operators. The authors note that a rigid interpretation of the definition of the outer-join does not yield the expected result when more than two outer-joins are performed in sequence, and introduce operators to correct this problem. A graph of dependencies among the input relations is assigned to each n-ary operator node to allow transformations that, under certain conditions:

1. transform outer-joins into joins
2. split n-ary outer-join nodes into an equivalent tree of two outer join nodes
3. distribute GADs over outer-joins
4. commute selections and projections over with GAD and outer-joins
5. distribute joins over outer-joins

Some of these transformations are similar to or extend transformations described in other approaches (e.g. 5 in [17], 1 and 4 in [14], and [54, 86], etc.). The application of these transformations is subject to conditions defined over the attributes involved in the transformed nodes. It is not clear how this framework will perform in practice. No cost model is defined to evaluate the benefits of the transformations and/or heuristics to determine which transformation is beneficial for a given tree, or how to choose a transformation

that will lead to the best tree. The framework has not been experimentally evaluated.

For other research on optimization of sequences of outer-joins the reader is referred to [28, 29] where outer joins are treated as disjunctions of joins and anti-semi-joins (as in AMOS*II*); [2] uses hypergraphs for outer-join reordering; and [32] describes how to push selections through outer-joins.

7.2 Object-oriented views

The integration facilities of AMOS*II* are based on work in the area of OO views [1, 37, 66, 68, 76, 46, 4, 68, 55]. This section presents a brief overview of two prototypes that have been most influential for the design of the OO views for data integration in AMOS*II*.

7.2.1 Multiview

The Multiview [46, 47, 66] OO view system adds dynamically updateable materialized OO views on the top of the GemStone OO DBMS. The views are defined by defining *virtual classes*, placed in the same class hierarchy as as the ordinary GemStone classes. The virtual classes are *capacity-augmenting*, i.e. attributes and methods can be added to them, as to the ordinary GemStone classes. Virtual classes are defined using six object-preserving algebra-operators:

- **select**: Returns a subset of the input class based on a predicate expression.
- **hide**: Removes properties from a set of objects.
- **refine**: Casts a set of input objects downwards in the class hierarchy (i.e. changes the class of the input objects to a subclass of their original class).
- **union**: Makes a union of two input class extents. The equality condition is OID equality.
- **intersection**: Returns the intersection of the extents of two classes.
- **difference**: Returns the difference of the extents of two classes.

The classes in GemStone and Multiview are organized in a multiple inheritance hierarchy. As AMOS*II*, GemStone also requires that each object instance belongs to a single most specific class. In presence of declaratively specified virtual classes, it is impossible to guarantee that two virtual classes will not both contain the same instance of some of their common superclasses, that at the same time is not an instance of any of their common subclassess. For example, if a class *Person* is subclassed by two virtual classes *Student* and *Teacher* having no common subclasses, there might be a person that satisfies both the conditions of being a student and a teacher. Such an instance would violate the requirements of belonging to a single most specific class. In [47] it is suggested that, to solve this problem, in a multiple inheritance OO views hierarchy the system must either generate automatically the intersection classes to classify this instances, or assign unique OIDs to the instances of the virtual classes. The second solution, applied in both Multibase and AMOS*II*, furthermore requires a *single point of inheritance* property of the class hierarchy. This property guarantees that two classes having inherited the same property, inherit it from a single class in the class hierarchy.

The Multibase system uses an elaborate solution where each object is represented by a single *conceptual object* and a number of *implementation objects* for each of its superclasses. The graph of each conceptual object with its corresponding implementation objects mirrors the class hierarchy.

This idea has been simplified and adapted in AMOS*II* where there is no distinction between implementation and conceptual objects. In AMOS*II*, these relationships are stored in coercion tables. The benefit of this approach is that, in a data integration scenario, new view classes can be defined over already existing populated classes. The instances of the view classes can then have their own OIDs without affecting the classes they are derived from.

Within the Multibase system, a class restructuring strategy is proposed to avoid conflicts in the class hierarchy by preserving the single point of inheritance property when new view classes are added. Because of the complexity of this process, in AMOS*II* we adopted some modeling constraints in order to prevent situations in which these transformations are needed.

Multiview is an implemented system with experimental results reported. However, it assumes active view materialization techniques and does not elaborate the consequences of the use of the applied techniques for data integration in a distributed heterogeneous environment.

7.2.2 O2 Views

The O_2 system is one of the first commercial OO systems to provide OO view functionality [57, 68]. Before the introduction of the OO view system, the O_2 system relied on named sets to provide some of the OO view features. Named sets however, do not provide some important features as: (i) description of the structure of the objects in the set, (ii) inheritance of methods from already defined classes (iii) attachment of new methods, etc.

The O_2 views are implemented on the top of the O_2 system. The views are defined using *virtual schemas* derived from *root schemas*. A root schema can either be another virtual schema or an O_2 schema. This allows for composition of views to an arbitrary degree of nesting. Corresponding to the root and virtual schemas there are a root and a virtual (data)base, representing the instances involved in the view mapping.

The views filter the data of the root base into the virtual base. Two modeling constructs are added to the O_2 data definition language to support the definitions of the filter mapping: *virtual classes* (VC) and *imaginary classes* (IC). A virtual class is defined as a subclass of a virtual or an ordinary O_2 class, named *root class*. A VC inherits the attributes of its root class, and can also have *virtual attributes* with functionality equivalent to the derived functions in AMOS*II*. Some attributes of the root class can be declared *hidden* and therefore not accessible to the user of the VC. Other properties of the VCs are that they:

- have an extent selected by a declarative query form the root database.
- are connected to the class hierarchy.
- provide a named set representing the extent.
- provide OIDs for the class instances based on the one-to-one correspondence with instances in the root database.

The ICs have the following properties:

- an extent is selected by a declarative query from the root database.
- they are not connected to the class hierarchy.
- assign OIDs to the instances based on a set of *core attributes*, corresponding to keys.

The following example, in which a VC *Adult* is defined as a specialization of the class *Person*, illustrates the language constructs used for the VC definition:

```
virtual class Adult from Person extension Adult
   virtual attributes
      age: integer has value self-> age;
   hide attribute date_of_birth
   includes
   (select p from p in People where p->age >= 21)
```

where *self* references the corresponding object of class *Person*, and the *includes* clause defines how is the extent of the VC selected from the extent of the root class.

The separation of the view definition facilities between the VC and IC constructs provide for a wide range of restructuring capabilities, while preserving the consistency of the class hierarchy. In comparison with AMOS*II*, the IC approach in AMOS*II* is used in the proxy types that retrieve their data from data sources other than AMOS*II* mediators. The VCs are equivalent to AMOS*II* DTs having a single supertype. In the query processing, AMOS*II* relies as much as possible on OIDs rather than on key values as the O_2 view system. When subtyping among AMOS*II* mediators, OIDs are used and manipulated because they are at least as small as the shortest possible key of an object. We assume that there is a functional dependency between the keys and the OID of an object, and therefore key manipulation is not needed in intersection-based OO views, such as the DTs.

The O_2 views mechanism does not provide multiple inheritance and integration facilities such as the DTs and IUTs in AMOS*II*. Therefore this approach can be classified as a class restructuring mechanism, or a selection-based view mechanism. For more advanced view definitions, the user is still limited to the named sets constructs.

Chapter 8

Summary and Conclusions

As a legacy of the mainframe computing trend in the previous decades, large enterprises often have many isolated data repositories used only within portions of the organization. While these systems contributed to the development of the companies in the past, their inability to interoperate and provide the user with a unified informational picture of whole enterprise is a "speed bump" in taking the corporate structures to the next level of efficiency. The recent development of the network technology bridged the physical gap between these systems, but nevertheless did not eliminate the burden of accessing the data in many diverse native formats.

Several technical obstacles arise in the design and implementation of data integration systems that provide the user with a unified view of data in multiple repositories (data sources). First, due to the distribution of the repositories, such a system has to operate in a distributed environment. Second, the data sources might use different data models and languages, and might contain equivalent, conflicting or complementary data, requiring reconciliation before it is presented to the user. Finally, the repositories are not under control of the data integration system, and their integration should not affect their functionality or require modifications.

The *wrapper-mediator* approach introduced in [85], divides the functionality of a data integration system into two units. The wrappers provides access to the data in the data sources using a *common data model* (CDM), and a *common query language*. The mediator provides a coherent view of the

data in the repositories by performing semantic reconciliation of the CDM data representations provided by the wrappers.

This thesis presents a design, implementation and evaluation of a mediator system named AMOS*II*. The mediation facilities in AMOS*II* are based on a passive approach where the requested data is retrieved from the data sources when a query is issued in the mediator. The passive approach preserves the autonomy of the data sources and is suitable for mediation in environments where data sources are autonomous, non-active, have large data volumes, or have high update frequencies. AMOS*II* is divided into two functional units:

- a mediation OO view mechanism providing constructs for reconciliation of data and schema heterogeneities among the sources.
- a multidatabase query processing engine for processing and executing queries over data in several AMOS*II* servers and other types of data sources.

The OO views mechanism is integrated in the inheritance mechanism by introducing derived types (DTs) and integration union types (IUT). The DTs and the UITs are placed in the same type hierarchy as the ordinary types.

The DT instances are derived from the instances of their supertypes according to a declarative condition specified in the DT definitions. DT instances are assigned OIDs, allowing their use in locally stored attributes defined over the DTs in the same way as over the ordinary types. Queries over DTs are expanded by system-inserted predicates that perform the DT system support tasks. The system support of the DT is divided into three mechanisms: (i) providing consistency of queries over DTs; (ii) generation of OIDs for the DT instances; and (iii) validation of the DT instances with assigned OIDs. The system generates templates and functions to perform these tasks. During the calculus generation phase, the query is analyzed, and where needed, the appropriate functions/templates are inserted. The final calculus representation is generated by a series of transformations aimed to produce a correct and efficient query calculus expression. In these transformations, query consistency is achieved by extent template expansions and removals, and by optimized coercion of local DT OIDs; OID generation is performed by including OID generation functions for selected query variables; DT instance validation is performed by inserting and expanding the

validation functions. The separation of the validation from extent generation (instance composition) leads to smaller validation functions. The separation of the OID generation from the extent generation allows selective generation of OIDs for the DT instances. Only the required portions of the DT extents are materialized locally.

The functions specifying the view support tasks describe relationships of the DTs in the type hierarchy and often have overlapping parts. The thesis demonstrates how calculus-based query optimization can be used to remove redundant computations introduced from the overlap among the system-inserted expressions, and between the system-inserted and user-specified parts of the query. The calculus-based transformations and optimizations do not require cost calculations and search space transitions, thus making them simple to implement and inexpensive to perform.

A novel framework for integration of data sources with overlapping data based on OO type hierarchies and late binding is presented. The IUTs are introduced to model a coherent view of heterogeneous data in multiple repositories. IUTs allows for resolutions of conflicts in the meta-data (e.g. naming, scaling, etc.) and for dealing with overlaps in the extents of the integrated types. Furthermore, instances of the IUTs can be assigned OIDs used in locally stored and derived functions.

Each IUT is mapped by the system to a hierarchy of system generated derived types, called auxiliary types (ATs). The ATs represent disjoint parts (a join and two anti-semi-joins) of the outer-join needed for the data integration. The reconciliation of the attributes of the integrated types is modeled by a system-generated set of overloaded derived functions, The implementation of each function is inferred from the *CASE* clause in the IUT definition.

Several novel query processing and optimization technique are developed for efficiently processing queries containing overloaded functions over the system-generated OO views. Queries over such an OO view hierarchy contain late-bound calls. The late-bound calls are translated to disjunctive calculus expressions that are suitable for application of techniques such as: bulk-oriented processing, type-aware query rewriting, selective OID generation, and dynamic generation of indexes for nested subqueries. The reported measurements compare the impacts of different query processing strategies showing that the combination of these techniques drastically lowers execution times, in some cases by several orders of magnitude.

The distributed mediation architecture of AMOS*II* is reflected in the design of the multidatabase query engine that processes queries over the

integrated OO views. It supports the cooperation of a number of AMOS*II* servers on a query processor level. An AMOS*II* system does not treat another AMOS*II* system as just another data source. More specifically, the inter-AMOS*II* interaction differs from the interaction between an AMOS*II* system and a wrapper in two main points:

- an AMOS*II* system can accept compilation and execution requests for subqueries over data in more than one data source. The wrapper interfaces accept subqueries that are always over data in a single data source.
- AMOS*II* supports materialization of intermediate results to be used as input to locally executed subqueries, generated by a query decomposition in another AMOS*II* server (*ship-and-execute* interface). A wrapper has only *execute* interface.

These two features influence the design of both the query decomposer and the run-time support for query execution. Techniques based on these features are used in AMOS*II* to achieve improved query performance.

The following conclusions can be drawn: First, although traditional object orientation allows for mediation by some remote method invocation protocol, its performance can be unacceptable. There is an apparent need for set-oriented query processing as used in the relational databases. Second, the multidatabase environment requires even greater optimization efforts to achieve acceptable performance for a wide range of queries. Third, describing type hierarchies and semantic heterogeneity using declarative functions and a functional CDM provides many opportunities for the extensive query optimization needed in an OO mediation framework.

The AMOS*II* system is implemented on a Windows NT/95 platform using TCP/IP for the communication.

Appendix A

Abbreviations

AT	auxiliary type
ATM	Asynchronous Transfer Mode
BS	bulk size
CDM	common data model
CQL	common query language
DBMS	database management system
DcT	decomposition tree
DST	data source type
DT	derived type
DTR	dynamic type resolver
ET	extent template
FMS	federated multidatabase systems
IS	input variables set (the set of input variables to a SF)
ISDN	Integrate Services Digital Network
IUT	integration union types
KS	a set of variables used at a remote SF (in SAE operator execution)
LAN	local area network
MDBMS	multidatabase management system
MIF	multiple implementation functions
NB	number of bulks

ODBC	Open DataBase Connectivity (standard)
ODMG	Object Database Management Group (consortium)
OID	object identifier
OO	object-oriented
PCA	project-concatenation algorithm
PPL	post processing list
QEP	query execution plan
RPC	remote procedure call
RS	result set (the set of variables returned by an SAE operator)
SAE	ship and execute
SAEDS	ship and execute operator description structure
SF	subquery function
SJA	semi-join algorithm
SJMA	semi-join with materialized index algorithm
SQL	Structured Query Language
SV	substitute variable
WCDMA	Wireless Collision Detection Media Access

References

[1] S. Abiteboul and A. Bonner: Objects and Views. *ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'91)*, pp. 238-247, ACM Press, 1991.

[2] G. Bhargva, P. Goel and B. Iyer: Hypergraph based reorderings of outer-join queries with complex predicates *ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'95)*, pp. 304-315, ACM Press, 1995.

[3] P. Bernstein and D. Chiu: Using Semi-joins to Solve Relational Queries. *Journal of ACM* Vol. 28, No. 1, pp. 25-40, 1981

[4] E. Bertino: A View Mechanism for Object-Oriented Databases. *3rd Intl. Conf. on Extending Database Technology (EDBT'92)*, Vienna, Austria, 1992.

[5] A. Bouguettaya, B. Benatallah and A. Elmagarmid: Interconnecting Heterogeneous Information Systems. Kluwer Academic Pulishers, The Netherlands, 1998.

[6] Silvio Brandani: Multi-database Access from Amos II using ODBC. In *Linköping Electronic Press*, Vol. 3, Nr. 19, Dec. 8th, 1998, *http://www.ep.liu.se/ea/cis/1998/019/.*

[7] O. Bukhres, A. Elmagarmid (eds.): Object-Oriented Multidatabase Systems, Prentice Hall, Englewood Cliffs, NJ, 1996.

[8] M. Carey, L. Haas, J. Kleewein and B. Reinwald: Data Access Interoperability in the IBM Database Family, *IEEE Data Engineering* 21(3), pp. 4-11, Sept. 1998.

[9] R. Cattell: The Object Database Standard: ODMG-93 2.0, Morgan Kaufman Publishers, San Mateo, CA, 1996

[10] E. Codd: A Relational Model for Large Shared Data Banks. *Communications of the ACM* Vol 13, No. 3, pp. 377-387, June 1970.

[11] U. Dayal, N. Goodman, T. Landers, K. Olson, J. M. Smith and L. Yedwab: Local Query Optimization in MULTIBASE: A System for Heterogeneous Distributed Databases, Technical Report CCA-81-11, Computer Corporation of America, 1981.

[12] U. Dayal, T. Landers and L. Yedwab: Global Query Optimization in Multibase: A System for Heterogeneous Distributed Databases, Technical Report CCA-82-05, Computer Corporation of America, 1982.

[13] U. Dayal: Processing Queries Over Generalization Hierarchies in a Mutltidatabase System, *9th Conf. on Very Large Databases (VLDB'83)*, Florence, Italy, 1983.

[14] U. Dayal, H. Hwang: View Definition and Generalization for Database Integration in a Multidatabase System, *IEEE Trans. on Software Eng.* 10(6), Nov. 1984.

[15] W. Du, R. Krishnamurthy and M-C. Shan: Query Optimization in Heterogeneous DBMS. *18th Conf. on Very Large Databases (VLDB'92)*, Vancouver, Canada, 1992.

[16] W. Du, M-C. Shan, J, Davis: Optimization and Execution Strategy for Multidatabase Queries. Techincal Report, Software Technology Laboratory HPL-94-74, April 1995.

[17] W. Du and M. Shan: Query Processing in Pegasus, *Object-Oriented Multidatabase Systems*, O. Bukhres, A. Elmagarmid (eds.), Prentice Hall, Englewood Cliffs, NJ, 1996.

[18] R. Elmasri and S. Navathe: Fundamentals of Database Systems. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[19] C. Evrendilek, A. Dogac, S. Nural, F. Ozcan: Query Optimization in Multidatabase Systems. *Journal of Distributed and Parallel Databases* Vol. 5 No. 1, pp. 77-114. January 1997.

[20] B. Finance, V. Smahi J. Fessy: Query Processing in IRO-DB, *Int. Conf. on Deductive and Object-Oriented Databases (DOOD'95)* pp. 299-319, 1995.

[21] D. Fishman, D. Beech, J. Annevelink, E. Chow, T. Connors, J. Davis, W. Hasan, C. Hoch, W. Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M-A. Neimat, T. Risch, M-C Shan, W. Wilkinson: Overview of the Iris DBMS. In W. Kim and F. Lochovsky (eds.): *Research Foundations in OO and Semantic DBS* pp. 174-199, 1990.

[22] G. Fahl, T. Risch, M. Sköld: AMOS - An Architecture for Active Mediators. *Workshop on Next Generation Information Technologies and Systems (NGITS'93)*, Haifa, Israel, June 1993.

[23] G. Fahl, T. Risch: Query Processing over Object Views of Relational Data. *The VLDB Journal*, 6(4), pp. 261-281, November 1997.

[24] D. Fang, S. Ghandeharizadeh, D. McLeod and A. Si: The Design, Implementation, and Evaluation of an Object-Based Sharing Mechanism for Federated Database System. *9th Intl. Conf. on Data Engineering (ICDE'93)*, (IEEE), Vienna, Austria, 1993.

[25] P. Fankhauser, G. Gardarin, M. Lopez, J. Munoz and A. Tomasic: Experiences in Federated Databases: From IRO-DB to MIRO-Web. *24st Conf. on Very Large Databases (VLDB'98)*, New York, NY, 1998.

[26] S. Flodin, T. Risch: Processing Object-Oriented Queries with Invertible Late Bound Functions, *21st Conf. on Very Large Databases (VLDB'95)*, Zurich, Switzerland, 1995.

[27] S. Flodin, V. Josifovski, T. Risch, M. Sköld and M. Werner: AMOS*II* User's Guide, available at *http://www.ida.liu.se/~edslab*.

[28] C. Galindo-Legaria: Outerjoins as Disjunctions, *ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'94)*, pp. 348-358, ACM Press, 1994.

[29] C. Galindo-Legaria and A. Rosenthal: Outerjoin Simplification and Reordering for Query Optimization, *ACM Transactions on Database Systems*, Vol. 22, No. 1, March 1997.

[30] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y.Sagiv, J. Ullman, V. Vassalos, J. Widom: The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems (JIIS)* Vol 8 No. 2 117-132, Kluwer Academic Pulishers, The Netherlands, 1997.

[31] M. Garcia-Solaco, F. Saltor, M. Castellanos: Semantic Heterogeneity in Multidatabase Systems, In O. Bukhres, A. Elmagarmid (eds.): *Object-Oriented Multidatabase Systems*, Prentice Hall, Englewood Cliffs, NJ, 1996.

[32] P. Goel and B. Iyer: Query Optimization: Reordering for a general Class of Queries. *ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'96)*, pp. 47-55, ACM Press, 1996.

[33] S. Grufman, F. Samson, S.M. Embury, P.M.D. Gray, T. Risch: Distributing Semantic Constraints Between Heterogeneous Databases. *13th International Conf. on Data Engineering (ICDE'97)*, (IEEE), Birmingham, England, 1997.

[34] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio and Y. Zhuge: The Stanford Data Warehousing Project, *IEEE Data Engineering*, 18(2), pp. 40-48, June 1995.

[35] L. Haas, D. Kossmann, E. Wimmers, J. Yang: An Optimizer for Heterogeneous Systems with NonStandard Data and Search Capabilities. *Data Engineering Bulletin* Vol. 19 No. 4 pp. 37-44, 1996.

[36] L. Haas, D. Kossmann, E. Wimmers, J. Yang: Optimizing Queries across Diverse Data Sources. *23th Int. Conf. on Very Large Databases (VLDB97)*, pp. 276-285, Athens Greece, 1997.

[37] S. Heiler and S. Zdonik: Object views: Extending the Vision. *6th International Conf. on Data Engineering (ICDE'90)*, IEEE, pp. 86-93, 1990.

[38] E. Horowitz, S. Sahni and D. Mehta: Fundamentals of Data Structures in C++. W H Freeman & Co., 1995.

[39] A. Hurson and M. Bright: Object-Oriented Multidatabase Systems. *Object-Oriented Multidatabase Systems*, O. Bukhres, A. Elmagarmid (eds.), Prentice Hall, Englewood Cliffs, NJ, 1996.

[40] IDC. Survey of 100 MIS Managers at Fortune 500 Companies. International Data Corporation, 1991.

[41] V. Josifovski and T. Risch: Calculus-based Transformations of Queries over Object-Oriented Views in a Database Mediator System, *3rd IFCIS International Conf. on Cooperative Information Systems*, New York City, August 1998.

[42] V. Josifovski and T. Risch: Functional Query Optimization over Object-Oriented Views for Data Integration *Journal of Intelligent Information Systems (JIIS)* Vol 12 No. 2/3, Kluwer Academic Pulishers, The Netherlands, 1999.

[43] W. Kelley, S. Gala, W. Kim, T. Reyes, B. Graham: Schema Architecture of the UNISQL/M Multidatabase System, *Modern Database Systems - The Object Model, Interoperability, and Beyond*, W. Kim (ed.), ACM Press, New York, NY, 1995.

[44] W. Kim, Y. Choi, S. Gala and M. Scheevel: On Resolving Semantic Heterogeneity in Multidatabase Systems. *Journal of Distributed and Parallel Databases* Vol 1. No. 3, pp. 251-279, July 1993.

[45] W. Kim and W. Kelley: On View Support in Object-Oriented Database Systems, *Modern Database Systems - The Object Model, Interoperability, and Beyond*, W. Kim (ed.), ACM Press/ Addison-Wesley Publishing Company, New York, NY, 1995.

[46] H. Kuno, Y. Ra and E. Rundensteiner: *The Object-Slicing Technique: A Flexible Object Representation and Its Evaluation,* Univ. of Michigan Tech. Report CSE-TR-241-95, 1995.

[47] H. Kuno and E. Rundensteiner: The MultiView OODB View System: Design and Implementation *University of Michigan Technical Report* CSE-TR-246-95, 1995.

[48] E-P. Lim, S-Y. Hwang, J. Srivastava, D. Clements, M. Ganesh: Myriad: Design and Implementation of a Federated Database System. *Software - Practice and Experience*, Vol. 25(5), 553-562, John Wiley & Sons, May 1995.

[49] E-P. Lim, J. Srivastava and S-Y. Hwang: An Algebraic Transformation Framework for Multidatabase Queries, *Journal of Distributed and Parallel Databases* Vol 3. No.3, pp. 273-307, Kluwer Academic Pulishers, The Netherlands, 1995.

[50] L. Liu and Calton Pu: An Adaptive Object-Oriented Approach to Integration and Access of Heterogeneous Information Sources. *Journal of Distributed and Parallel Databases* Vol 5. No. 2, pp. 167-205, Kluwer Academic Pulishers, The Netherlands, 1997.

[51] W. Litwin, L. Mark and N. Rossopoulos: Interoperability of Multiple Autonomous Databases. *ACM Computing Surveys*, Vol 22. No. 3 pp. 267-293, 1990.

[52] W. Litwin and T. Risch: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates. *IEEE Transactions on Knowledge and Data Engineering* 4(6), pp. 517-528, 1992.

[53] P. Lyngbaek et al: *OSQL: A Language for Object Databases*, Tech. Report, HP Labs, HPL-DTD-91-4, 1991.

[54] W. Meng, K-L. Liu and C. Yu: Query Decomposition in Multidatabase Systems. Technical Report CS-TR-93-9, Department of Computer Science, State University of New York and Binghampton, 1993.

[55] A. Motro: Superviews: Virtual Integration of Multiple Databases. *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 7, July 1987.

[56] S. Nural, P. Koksal, F. Ozcan, A. Dogac: Query Decomposition and Processing in Multidatabase Systems. *OODBMS Symposium of the European Joint Conference on Engineering Systems Design and Analysis*, Montpellier, July 1996.

[57] O2 Technology: O2 Views User Manual, version 1, Dec. 1993.

[58] Object Management Group: *The Common Object Request Broker: Architecture and Specification*, Object Request Broker Task Force, 1993.

[59] K. Orsborn and T. Risch: Next Generation of O-O Database Techniques in Finite Element Analysis. *The Third International Conf. on Computational Structures Technology*, Budapest, Hungary, August 21-23, 1996.

[60] F. Ozcan, S. Nural, P. Koksal, C. Evrendilek, A. Dogac: Dynamic Query Optimization on a Distributed Object Management Platform *Fifth International Conference on Information and Knowledge Management (CIKM96)*, Maryland, USA, November 1996.

[61] M. T. Ozsu and P. Valdurez: Principles of Distributed Database Systems (2nd edition), Prentice-Hall International Inc., London UK, 1999.

[62] Y. Papakonstantinou H. Garcia-Molina, J. Widom: Object Exchange Across Heterogeneous Information Sources, *The Eleventh Intl. Conf on Data Engineering (ICDE95)*, Taipei, Taiwan, 1995.

[63] K. Richine: Distributed Query Scheduling in DIOM. Tech. Report TR97-03, Computer Science Department, University of Alberta, 1997.

[64] M. Roth and P. Schwarz: Don't Scrap It, Wrap It. *23th Int. Conf. on Very Large Databases (VLDB97)*, pp. 266-275, Athens Greece, 1997.

[65] F. Rendeze, K. Hergula: The Heterogeneity Problem and Middleware Technology: Experience and performance of database gateways. *24th Conf. on Very Large Databases (VLDB'98)*, New York, 1998.

[66] E. Rundensteiner, H. Kuno, Y. Ra, V. Crestana-Taube, M. Jones and P. Marron The MultiView project: object-oriented view technology and applications, *ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'96)*, pp. 555-563, ACM Press, 1996.

[67] C. Santos: *Design and Implementation of an Object-Oriented View Mechanism,* GOODSTEP ESPRIT-III Technical Report, ESPRIT-III Project No. 6115, 1994.

[68] C. Souza dos Santos, S. Abiteboul and C. Delobel: Virtual Schemas and Bases. *4th Intl. Conf. on Extending Database Technology (EDBT'92)*, Viena, Austria, 1992.

[69] A. Sheth and J. Larson: Federated database systems and managing distributed, heterogeneous and autonomous databases. *ACM Transactions on Database Systems* Vol 6. No. 1, pp. 140-173, 1990.

[70] A. Sheth and V. Kashyap: So far (Schematically) yet So Near (Semantically) *IFIP WG 2.6 Conference on Semantics of Interoperable Databases* (Data Semantics 5), North Holand, Amsterdam 1993. pp. 283-312

[71] D. Shipman: The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1), ACM Press, 1981.

[72] A. Silberschatz, H. Korth and S. Sudarshan: Database System Concepts 3rd ed., McGraw Hill, New York, 1997.

[73] V. Smahi, J. Fessy and B. Finance: Query Processing in IRO-DB Technical Report PRiSM, Versailles University 1994/37, Versailles, France, 1994.

[74] ISO and ANSI SQL Working Draft, X3H2-93-359, August 1993.

[75] M. Sköld, T. Risch: Using Partial Differencing for Efficient Monitoring of Deferred Complex Rule Conditions. *12th International Conf. on Data Engineering (ICDE'96)*, (IEEE), New Orleans, Louisiana, Feb. 1996.

[76] M. Scholl, C. Laasch and M. Tresch: Updatable Views in Object-Oriented Databases. *Second Deductive and Object-Oriented Databases Conference (DOOD91)*, Dec, 1991.

[77] D. Straube, T. Özsu: Query Optimization and Execution Plan Generation in Object-Oriented Database Systems. *IEEE Transactions on Knowledge and Data Engineering* 7(2), pp. 210-227, 1995.

[78] S. Subramananian and S. Venkataraman: Cost-Based Optimization of Decision Support Queries using Transient Views. *ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'98)*, pp. 329 - 330, 1998.

[79] A. Tanenbaum: Computer Networks. Prentice Hall, Englewood Cliffs, NJ, 1996.

[80] M. Templeto, D. Brill, A. Chen, S. Dao, E.Lund, R. McGregor and P. Ward: Mermaid: a front end to distributed heterogeneous database. *Special Issue on Distributed Database, Proceedings of the IEEE 75*, pp. 695-708, May 1987.

[81] A. Tomasic, L. Raschid, P. Valduriez: Scaling Access to Heterogeneous Data Sources with DISCO. *Transactions on Knowledge and Data Engineering* (TKDE) vol. 10 No. 5, pp. 808-823, 1998.

[82] S. Venkataraman and T. Zhang: Heterogeneous Database Query Optimization in DB2 Universal DataJoiner *24th Conf. on Very Large Databases (VLDB'98)*, pp. 685 - 689, New York, 1998.

[83] J. Ullman and J. Widom: A First Course in Database Systems. Prentice Hall, Englewood Cliffs, NJ, 1998.

[84] M. Werner: Mutidatabase Integration using Polymorphic Queries and Views. Licenciate Thesis No. 546, Department of Computer and Information Science, Linköpings Universitet, Linköping, Sweden, 1996.

[85] G Wiederhold: Mediators in the Architecture of Future Information Systems, *IEEE Computer*, 25(3), Mar. 1992.

[86] C. Yu and W. Meng: Principles of Database Query Processing for Advanced Applications, Morgan Kaufman Publishers, San Francisco CA, 1998.

[87] G. Zhou, R. Hull, R. King and J. Franchitti, Data Integration and Warehousing Using H2O, *IEEE Data Engineering*, 18(2), pp. 29-40, June 1995.

[88] Q. Zhu and P. Larson: A Query Sampling Method for Estimating Local Cost Parameters in a Multidatabase System. *10th Intl. Conf. on Data Engineering (ICDE'94)*, (IEEE), pp. 144-153, Houston, Texas, 1994.

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology
Linköping Studies in Information Science. Dissertations

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall:** Semantisk analys av processbeskrivningar i naturligt språk, 1977, ISBN 91-7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91-7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91-7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation,1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372-805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies,1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani**: Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea / ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms,1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund**: Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld**: Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén**: Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Langugaes from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis,1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations.

No 582 **Vanja Josifovski:** Design, Implementation and Evaluation of A Distributed Mediator System for Data Integration.

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturering - att skapa samstämmighet mellan informationssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.

No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998. ISBN-9172-19-299-2.

Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration: the Story of AMOS*II*

Vanja Josifovski

An important factor of the strength of a modern enterprise is its capability to effectively store and process information. As a legacy of the mainframe computing trend in recent decades, large enterprises often have many isolated data repositories used only within portions of the organization. The methodology used in the development of such systems, also known as *legacy systems*, is tailored according to the application, without concern for the rest of the organization. From organizational reasons, such isolated systems still emerge within different portions of the enterprises. While these systems improve the efficiency of the individual enterprise units, their inability to interoperate and provide the user with a unified information picture of the whole enterprise is a "speed bump" in taking the corporate structures to the next level of efficiency.
Several technical obstacles arise in the design and implementation of a system for integration of such data repositories (sources), most notably distribution, autonomy, and data heterogeneity. This thesis presents a data integration system based on the wrapper-mediator approach. In particular, it describes the facilities for passive data mediation in the AMOS*II* system. These facilities consist of: (i) object-oriented (OO) database views for reconciliation of data and schema heterogeneities among the sources, and (ii) a multidatabase query processing engine for processing and executing of queries over data in several data sources with different processing capabilities. Some of the major data integration features of AMOS*II* are:

- A distributed mediator architecture where query plans are generated using a distributed compilation in several communicating mediator and wrapper servers.
- Data integration by reconciled OO views spanning over multiple mediators and specified through declarative OO queries. These views are *capacity augmenting* views, i.e. locally stored attributes can be associated with them.
- Processing and optimization of queries to the reconciled views using OO concepts such as overloading, late binding, and type-aware query rewrites.
- Query optimization strategies for efficient processing of queries over a combination of locally stored and reconciled data from external data sources.

The AMOS*II* system is implemented on a Windows NT/95 platform.

Vanja Josifovski is a researcher at the Department of Computer and Information Science at the Linköpings universitet, Linköping, Sweden. Previously he was a member of the Database Systems Research and Development Center at the University of Florida at Gainesville. His research interests include query processing and optimization, data integration and distributed database systems.