Linköping Electronic Articles in
computer and information science.
Vol. 3(1998): nr 19

Multi-database Access from Amos II using ODBC

Silvio Brandani

Department of Computer and Information Science
Linköping University
Linköping, Sweden

Linköping University Electronic Press
Linköping, Sweden

http://www.ep.liu.se/ea/cis/1998/019/

Published on November 27, 1998 by
Linköping University Electronic Press
581 83 Linköping

Linköping Electronic Articles in
computer and information science
ISSN 1401-9841
Series editor: Erik Sandewall

Recommended citation:
<Auther>. <Title>. Linköping electronic articles
in computer and information science, Vol. 3(1998): nr 19.
http://www.ep.liu.se/ea/cis/1998/019/.

This URL will also contain a link to the auther's home page.

The publishers will keep this line on the Internet
(or its possiarticle on- ble replacement network in the future)
for a period of 25 years from the date of publication,
barring exceptional circumstances as described separately.

ABSTRACT

In recent years, there has been a tremendous proliferation of databases in the work place, dominated by relational databases systems. An emerging need for sharing data across different database platforms has introduced the need for Multi-DataBase Systems (MDBS). MDBS are systems capable of operating over a network and encompassing a heterogeneous mix of different database systems, providing the user with a unified view of distributed and heterogeneous data.

The aim of this work is to present an approach to the problem of building high level abstractions as views that integrate data from different relational databases. The mechanism presented can serve as a basis for a multidatabase system.

As a platform for this work we use the object oriented multi-database system Amos II. To access data stored in different databases the ODBC standard is used. ODBC provide multidatabase connectivity using a common interface approach, which gives programmers a uniform SQL interface to different relational databases.

A number of problems were identified and alternative solutions were in most cases proposed and implemented. Measurements on the implementation are reported and discussed.

ACKNOWLEDGEMENTS

I would like to thank my teacher and supervisor Tore Risch for his support during the work on this final report. I would also like to thank all members of the Laboratory for Engineering Databases and Systems (EDSLAB).

I am also grateful to Vanja Josifovsky who helped me with the English language.

CONTENTS

1 INTRODUCTION 7

1.1 THE PROBLEM: WHY MULTIDATABASE ACCESS ? 7
1.2 BACKGROUND 8
1.2.1 Multi-database system 8
1.2.2 Object Oriented databases 9
1.2.3 Amos 9
1.2.4 ODBC 9
1.3 THE AIM 10
1.4 LIMITATIONS 10
1.5 REPORT OVERVIEW 11

2 BACKGROUND 12

2.1 RELATIONAL DATABASES 12
2.1.1 Standard Query Language (SQL) 14
2.1.2 Views 16
2.2 OBJECT ORIENTED DATABASES 17
2.3 OBJECT-ORIENTED MULTI-DATABASES 18
2.3.1 The new environment 18
2.3.2 The object-oriented multi-database solution 20
2.3.3 The global schema approach 21
2.3.4 The multidatabase language approach 22
2.3.5 Commercial standards 23
2.3.6 Object Oriented views 23
2.4 THE AMOS O-O MULTI-DATABASE SYSTEM 24
2.4.1 The AMOS multidatabase system architecture 24
2.4.2 The AMOS data model 25
2.4.3 The AMOS query language 26
2.5 ODBC 27

3 DESIGN 28

3.1 APPROACHES 28
3.2 ARCHITECTURE 29
3.3 PHASES 30

4 EXTENDING AMOSQL WITH ODBC ACCESS 31

4.1 EXAMPLE MULTIDATABASE 31
4.1.1 Scenario 31
4.1.2 Schema and data conflicts 32
4.2 ODBC DATA SOURCES IN AMOS 33

4.3 How to define an OSQL function that calls an arbitrary SQL statement 33
4.4 How to use the SQL function for multi-database views. 34

5 IMPLEMENTATION 37

5.1 ODBC 37
5.1.1 The ODBC architecture 37
5.1.2 The ODBC interface 39
5.2 Using streamed callin-callout interface in Amos II through ODBC 40
5.2.1 Principles 40
5.2.2 The driver program 41
5.2.3 Function implementation 42
5.3 Prepare optimization 45

6 PERFORMANCE MEASUREMENTS 47

6.1 Setup 47
6.2 The measurements 48

7 CONCLUSIONS AND FUTURE WORK 50

APPENDIX 51

REFERENCES 54

1 Introduction

This first chapter outline the issues addressed in this report. The task and delimitations are then discussed, and finally there is an overview of the report.

1.1 The problem: Why multidatabase access ?

Database applications and technologies are of central importance in many information systems. A database is a collection of related data. The collection of programs that enables users to create and maintain a database is called a database management system (DBMS).
Historically , companies used a single database system. It consisted of one or more databases managed under the centralized control of a *monolithic* database management system. All database access was done either through the front end of such systems or through applications written to work exclusively with such systems.

Due to historical reasons, and the fact that different applications can be better supported by different types of database systems, today, in a typical medium or large size enterprise many different kinds of database management systems are used. The databases are often *distributed* over a network of mainframe computer servers, workstations and personal computers. They are often *heterogeneous* due to the differences in the semantics of data and in DBMSs (different data models and query languages used). And, if they are managed independently by separate organizations, they are *autonomous*.

The monolithic approach is insufficient in this new environment where organizations require solutions to the following problems:

1. There are applications that need to access more than one database system. So, application programmers have to use various database interfaces and need to know the databases (locations) of required data.

2. Some data may be stored redundantly in many databases,

possibly in different representations. Hence, the risk of data inconsistency is huge and the risk for wrong decisions increases.

The area of multidatabase systems deals with the question of what approach should be used to access and merge data in different, existing information systems.

1.2 Background

This section briefly describes the principles and approaches to the problem discussed above and subjects related to our work. These concepts will be present in a greater detail in the next chapter.

1.2.1 Multi-database system

A *multidatabase system* acts as a front end system that allow the users to integrate information from preexisting, heterogeneous local databases.
The two major design approaches for multidatabases are the *global schema* approach and *multidatabase language* approach. They are based, respectively, on the idea of: re-engineering and interoperation.

The *re-engineering* approach transfer application logic, the data definition, and the data from the old systems into a new system: all the accessible databases are globally integrated. The advantage of the re-engineering approach is that the new systems are easier to maintain. However, this process is expensive and complicated and, in many companies, it will be impossible to integrate all these databases under a global schema.
The second option is to allow the existing systems to *interoperate* by providing a standard interface to the existing databases. A multidatabase language system puts most of the integration responsibility on user giving them many support functions for database interoperability.

A central problem in multidatabase systems is heterogeneity in the semantic of the data stored in various repositories. To handle this most systems use a *common data model*. With this approach the schemas of the participating databases are mapped to equivalent schemas in the common data model. These schemas can be seen as views of the underlying databases.
A *view* is a logical description of data. All operations against a view are translated into commands against the underlying schema according to some mapping between the view and the underlying schema. Furthermore views enable sharing data among information systems.

1.2.2 Object Oriented databases

O-O (Object Oriented) DBMSs combine the features of O-O programming languages with those of the DBMSs.
The advanced modeling capabilities of the object model offer the advantage to capture all the semantics of the participating databases when used in the context of data integration problem.
O-O databases play an important role; in fact the O-O paradigm appears to be the approach of choice for the common data model used to provide a homogeneous interface that handle the problem of access data in their diverse native formats. Such interface or view mechanism makes it possible for users to transparently work with data in a relational database as if it was stored in an O-O database.

1.2.3 Amos

AMOS [2] (Active Mediators Object System) is a object-relational research DBMS prototype. AMOS is an extension of a main-memory version of Iris[5], called WS-Iris [1]. AMOS provides an O-O model as the framework for uniform interoperation of multiple data sources.
AMOS includes the AMOSQL query language that provides a declarative query interface for defining, populating and manipulating the database.
AMOSQL is an extensible and O-O query language that subsumes the relational calculus. AMOSQL is derivative of OSQL[4] which is a functional and O-O language, originating from DAPLEX [3]. The central part of the Amos project has so far been multidatabase systems. In fact, one of the key capabilities being developed with in the Amos project is to be able to state O-O queries that combine data from several data sources such as relational databases [14], [15]. For this purpose the Amos multidatabase system architecture is based on the wrapper-mediator approach [11].

1.2.4 ODBC

In regards to the problem of data-interchange among various databases and applications, a group of leaders in the database and systems industry, called SQL Access Group (SAG), has defined a Call Level Interface (CLI) which provides a common interface for accessing relational databases using SQL. Based on this CLI, Microsoft proposed specifications of an API (Application Program Interface) called *ODBC* (Open Database Connectivity)[10]. ODBC is designed to be a superset of the SAG CLI (the ODBC extensions provide additional functionality).

1.3 The aim

This report addresses the problem of defining views that span over a few relational databases and provide the basic primitives to build an O-O view mechanism. Our approach combine the ODBC functionality to extract data with the Amos facilities to manipulate and combine the extracted data using its query and data modeling language AMOSQL.
This combination allows effective sharing of information from disparate database systems and classifies our work in the multidatabase language approach (with some differences that will be pointed out in the following chapters).

The primary goal is to implement SQL access from AMOSQL using the extensibility of Amos II. Extensibility is supported through foreign functions written in an external programming language(C or Lisp). In this way relational databases can be accessed from queries.

The secondary goal is to define multidatabase views accessing several relational databases. Multidatabase views hide the differences of the participating databases and provide a coherent interface to the users.
In Amos views are supported through derived functions that use the AMOSQL query language to calculate the extension.

1.4 Limitations

We do not attempt to create O-O views [14] over relational databases. In fact, to allow O-O view definitions, our system must be combined with other work on how to transform data expressed in the relational data models into an object oriented data model used as common data model (CDM). Another key issue when object views of relational databases are developed regard how to provide in the CDM a semantically coherent representation of the data in the relational databases that may contain heterogeneity in the semantic of data describing same real world entities.

1.5 Report overview

This report consist of seven chapters.
Chapter 2 presents a theoretical background to the area of interest, i.e. relational databases, SQL, views, object oriented databases, O-O multidatabase systems and Amos, ODBC.
Chapter 3 describes the design methods (approaches, architecture and phases) used during the implementation.
Chapter 4 presents the databases used throughout the examples, describe how to use ODBC data sources in Amos and how to extend Amosql with ODBC access.
Chapter 5 describes the implementation of the system we have built.
Chapter 6 presents the measurements done.
Chapter 7 gives the conclusions drawn from the implementation and benchmarking. It also discusses future work.

2 Background

This chapter presents the technical background to the area of interest.
First various concepts related to our task are presented, including
- relational databases
- SQL and the various ways that applications use it.
- Views

Then, we present the concept of object oriented databases in order to introduce the reader to the following, central sections on multidatabase systems and the Amos system.
Finally the concepts behind the development of ODBC are discussed.

2.1 Relational databases

A database is a collection of related data stored in a computer [8].
A DBMS is a collection of programs for creating, searching, updating and manipulating databases.
There are two kinds of information in a database, data and schema. The schema is metadata describing the semantic of the information in a given database. It is analogous to the data-type declarations in a programming language. Each DBMS supports one data model, that is the type of data abstraction used to provide a conceptual representation of data without many of the details of how the data is stored. The most common data model is the relational data model.

The relational data model was introduced to the database community in 1970 by E.F.Codd [13]. It support the separation of the user view of data from its physical implementation. The relational model is based on a single uniform data structure called a *relation* that is used to represent both data and interrelations. A relation is commonly viewed as a named *table*. The rows of the table, called tuples, are not ordered. The columns, called attributes, have a name and an associated data type.

A table can represent both a class or a relationship between classes. When the table represent a class, the tuples represent specific instances of that class and the attributes represent common properties for the class. Similarly, a table can be used to represent a relationship, whose

instances are represented by rows in the table. In this case, the columns represent the participating entity types.

To understand the relational model, we shall represent an employee application as relational tables. We will need two tables (relations) for our relational data schema, one for each kind of object in our application: EMPLOYEE and DEPARTMENT. Fig 2.1 illustrates these tables.
The figure shows some rows in the tables, representing individual employees and departments. The EMPID attribute of the EMPLOYEE table and the DEPTID attribute of the DEPARTMENT table are the primary keys, in fact their value is unique in every row. The DEPT attribute of the EMPLOYEE table is a foreign key that take on values of the primary key DEPTID of the DEPARTMENT table.

EMPLOYEE TABLE

EMPID	NAME	DEPT
5	Lambert, Kim	130
11	Weston, K. J.	130
15	Young, Katy	623
28	Bennet, Ann	120
29	De Souza, Roger	623
36	Reeves, Roger	120
37	Stansbury, Willie	120

DEPARTMENT TABLE

DEPTID	NAME	LOCATION
120	European Headquarters	London
130	Field Office: East Coast	Boston
623	Customer Support	Monterey

Figure 2.1

In the Fig 2.2 we show a relationship table, called MANAGER, used to represent associations between departments and employees. There is one row in the MANAGER relationship table for each unique department-employee pair.

MANAGER TABLE

DEPARTMENT	EMPLOYEE
European Headquarters	28
Field Office: East Coast	11

Customer Support	15

Figure 2.2

One of the most important ways to manipulate data in a relational database are through the declarative language SQL that has become a standard language for end-users and application programming interfaces.

2.1.1 Standard Query Language (SQL)

This section presents an introductory history of SQL , or Structured Query Language, and reasons why such a common language and a Call Level Interface(CLI) that supports SQL evolved.

Originally, the users of DBMSs were programmers. Accessing the stored data required writing a program in a programming language such as COBOL. While these programs were often written to present a relatively friendly interface to a non-technical user, access to the data itself required the services of a knowledgeable programmer. Ad hoc access to the data was not an easy and practical task**.**

Allowing users to access data on an ad-hoc basis required giving them a language in which to express their requests. A single request to a database is called a query; the language the request are formulated in is called a query language. One of the most popular query languages based on the relational data model is SQL (Structured Query Language).

SQL can be used to query, define, manipulate and control data in the database, it is the database application language that allows to use the relational model. Since the relational model is based on set theory, an SQL statement returns information in the tabular format of the relational model. This requires that SQL rely on concepts such as table, indexes, keys, rows and columns to access and present information.

The most important statement is the SELECT statement, used to retrieve data from the database, based on a declarative query condition.

For example, the DEPARTMENT table in figure 2.1 has a column NAME that contains the name of the department and a column DEPTID that contains the code of the department. The EMPLOYEE table also has a column NAME that specifies the name of the employee and a column DEPT that specifies the code of the employee's department. A query that returns the names of the employees in the Marketing department can be specified as following:

```
SELECT EMPLOYEE.NAME
FROM                EMPLOYEE,   DEPARTMENT
```

```
WHERE DEPARTMENT.NAME ='Marketing'
AND DEPT_ID =DEPT
```

A query language is not called relationally complete unless it provides at least three basic operations: selection, projection, and join. Selection produces a subset of the rows of a table; the preceding queries do this by specifying constrains on department names after the WHERE keyword. Projection produces a subset of the columns of a table; the preceding queries do this by specifying attribute names after the SELECT keyword. The join operation matches up records in two different tables that have equal or related values in specified attributes. The preceding query performs a simple join on the department identifier in the employee and department tables.

Because of the need for data access by computer programs and because of the tabular format used in SQL is very different from the record-at-a-time method used from most programming languages to manipulate data, a *call-level interface* (CLI) was developed to send SQL statements to the DBMS within program's code. A call-level interface provides a library of DBMS functions that can be called by the application program. Thus, instead of trying to blend SQL with another programming language, a call-level interface is similar to the routine libraries most programmers are accustomed using, such as the string, I/O, or math libraries in C.
Using a call-level interface typically involves the following steps:

1. The application calls a CLI function to connect to the DBMS.
2. The application builds an SQL statement and places it in a buffer. It then calls one or more CLI functions to send the statement to the DBMS for preparation and execution.
3. If the statement is a SELECT statement, the application calls a CLI function to return the results in an application buffers. Typically, this function returns one row or one column of data at a time.
4. The application calls a CLI function to disconnect from the DBMS.

2.1.2 Views

A database mechanism to obtain selected data for an application is a view specification. Views are an important subject in this report since the purpose of our work is to define views that span different relational databases or multidatabase views.

A view is defined in a relational model as a query over the base relations, and pheraps also over other views. The view, like any relational query result, is a relation and looks like a table, but no data is stored in a view. Instead the data of view is retrieved through its definition when it is needed.
Views are collocated in the third level in the **ANSI/SPARC** three-level architecture (Fig 2.3).

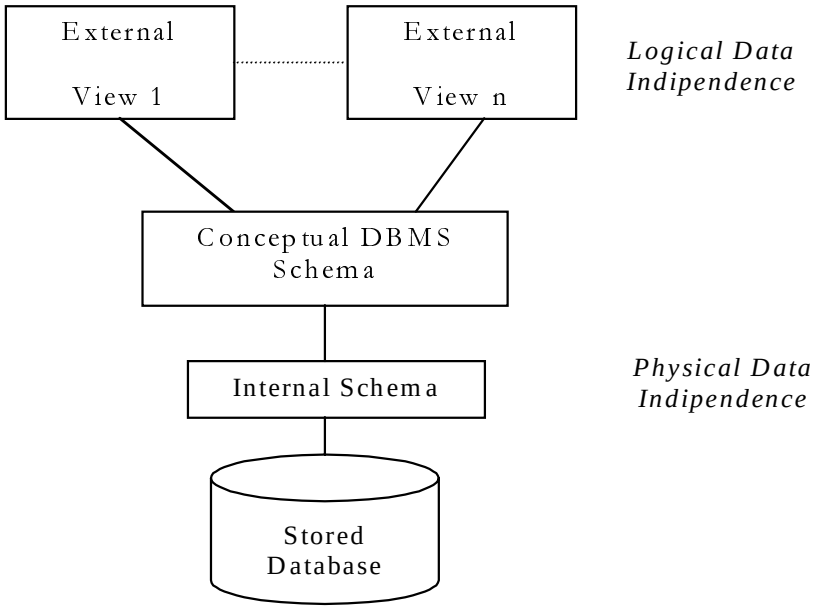

Figure 2.3 The ANSI/SPARC three-schema Architecture

The ANSI/SPARC is a well-known architecture for describing databases, its aims is to isolate the physical database from applications (data independence). At the first level there is the internal schema that describes the physical storage structure of the database. The middle level involves the conceptual schema which decribes the structure of the whole database. Finally, the third level represents the external schemas or views and describes subsets of the conceptual schema that are relevant to particular applications. The data model of the view is always the same as the data

model of the schema it is defined over.
Furthermore views provide logical data independence that is the capacity to change the conceptual schema (the representation of one or more tables) without having to change external schemas. The fact that data are not stored in a view is transparent from users.
Views are intended to provide a better level of abstraction and to reduce the complexity of the database design for its users.

2.2 Object Oriented databases

Object Oriented DBMSs are expected to meet the requirements of new application domains, such as computer-aided design, software engineering and office automation, which have a need for functionality beyond those provided by relational databases.
There is not a single accepted data model for O-O databases, however the following basic O-O *concepts* are supported in most of the OO DBMS:

The fundamental concept and modeling construct in an OO DBMS is the concept of *object*. Objects are used to model physical or abstract entity in the domain of interest that possesses certain characteristics (attributes). Objects have an existence which is independent of the values of their attributes.
Each object has a unique, immutable, system generated identifier, called object identifier(OID).
Properties of objects are represented with a set of attributes, their values define the *state* of the object. *Attributes* can be simple if they are used to describe properties that can be represented by literal values such as integer, reals or strings.
The value of an attribute may be another object in which case the attribute represent relationship between objects (reference attributes).
The *behavior* of the objects is defined by operations or *methods* which can be carried out over an object. In the pure OO systems all communication with objects is performed through methods that hide the internal structure of the objects. This is called *encapsulation* and provides a form of data independence; however in O-O databases encapsulation is often violated.

The structure and behavior of the objects (also called the intent) are defined by *types* that are also used for controlling the type extent, i.e. the set of objects that adhere to a certain type. All instances of a type have the same attributes and behavior. O-O concepts usually involve some *generalization* mechanism for types, providing capabilities to structure types into hierarchies. These hierarchies define sub- and super-type relationship among types. The subtypes *inherits* the attributes and behavior of the supertype and may have additional attributes and methods that the supertype does not have. When a type inherits from several parallel supertypes it is called multiple inheritance.

In the AMOS data model, functions are used to model both properties of objects, relationships between objects, and operations on objects.
Because of a need for a model which can encompass all different models and the richer modeling capabilities than the relational model, the object oriented model is suitable to facilitate data interoperability of heterogeneous databases.

There is currently no standardized specification for an object data model. However, there are two emerging standards, ODMG [6] and SQL3 [7], that define their own object data model in combination with query facilities.
A current trend is to extend relational database systems with object oriented concepts. These systems are usually called object-relational database systems. AMOS is based on this approach. An important aspect of OR database technology is the availability of several kind of extensibility including query language and query processing.
The AMOSQL query language, described in Section 2.4.1, will here exemplify the capabilities of an OO query language.

2.3 Object-Oriented multi-databases

In this section are presented the concepts behind the development of object-oriented multidatabase systems, their design approaches are discussed and is presented an overview of the state of the art for commercially available multidatabase systems. Finally the importance of the O-O views is described.

2.3.1 The new environment

Today, we can find many different database systems in use. This heterogeneity primarily has technical reasons since no database system is general enough to be appropriate to different data management requirements. Moreover, it has economical reasons because there are various database systems on the market suitable for the same application domain which differ in service support and price. In general, also organizational reasons exist due to company reorganizations and mergers, integrating information across departments where departments that previously had a single DBMS now has several.

The basic property of such databases is that they are independently created and administered . This independence or local autonomy implies that local data models may be different. Even if the databases all use the same data model, data definition autonomy still allow for mutual semantic conflicts, as for example heterogeneity of names, value types and meaning among similar data. They result from different perceptions of the same reality by different people.
In this report, we limit to the problem of integration of

heterogeneous data represented in a single data model, the relational one.

Figure 2.4. illustrates the situation we need to be able to handle. We have a number of databases distributed over a computer network. Each of these databases are managed by some DBMS and each of them are used by one or more applications. The database systems are heterogeneous and autonomous. To this we want to add a number of applications that can access data from several of these component databases.

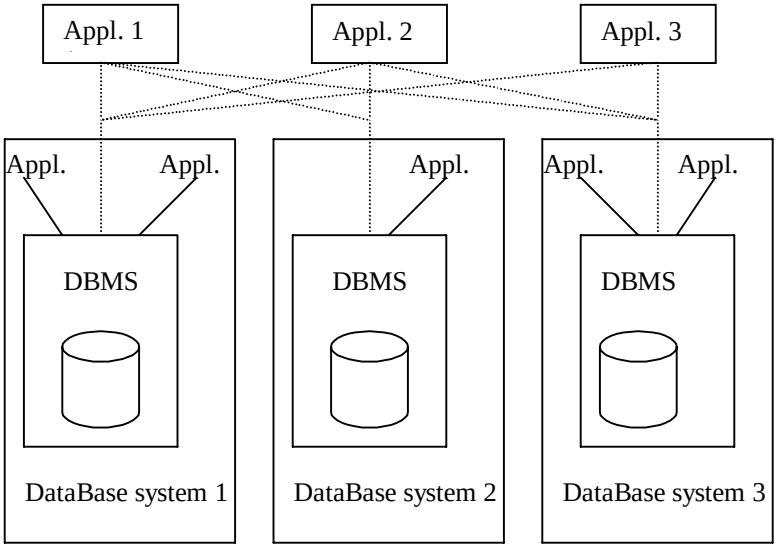

Figure 2.4. Applications accessing several existing databases.

The issue is critically important because data are at the heart of almost any computer system and the efficient and effective use of information provides business values and strategic advantage.
The lack of effective data sharing causes inefficient engineering and manufacturing activities and business operations.

2.3.2 The object-oriented multi-database solution

One solution for data integration is to construct a front end system that supports a single common data model (to handle the problem of heterogeneity in the semantic of data) and a single global query language on top of different types of existing databases. The front end system plus the underlying database systems is called a *multidatabase system*.
The front end system used in our task is the Amos O-O multidatabase system, presented in the next section.
It is important to develop a technology that can deal with the database autonomy. As noted in the previous sections, the object-oriented technology is a primary choice for this task. We discussed its principles in the context of DBMSs in section 2.2. Here we only comment on its role in addressing the interoperability issues.

Object-oriented systems typically treat the entities in their domain as instances of some abstract type. In the case of database systems, these entities are usually data objects. The technology is primarily concerned with providing the necessary tools and methodology for the consistent management of these entities while there are user accesses to them. This is to be done without allowing the user to see the physical structure of each object. In other words, user access to objects is only by means of well-defined interfaces which hide their internal structure. In the case of applying object-orientation to the interoperability problem, one considers each autonomous DBMS to be an object. The relationship is then established between manipulating objects without 'looking inside them' and being able to perform operations on autonomous DBMSs without 'requiring changes in them'. This is a very simple characterization of the problem.

The main alternatives for multidatabase systems architectures [9] have all in common that users access the component databases through different kinds of non-materialized views.
The differences between the architectures lies in what kinds of views are provided. An overview of the schema architectures of the different approaches is given in Fig. 2.4.

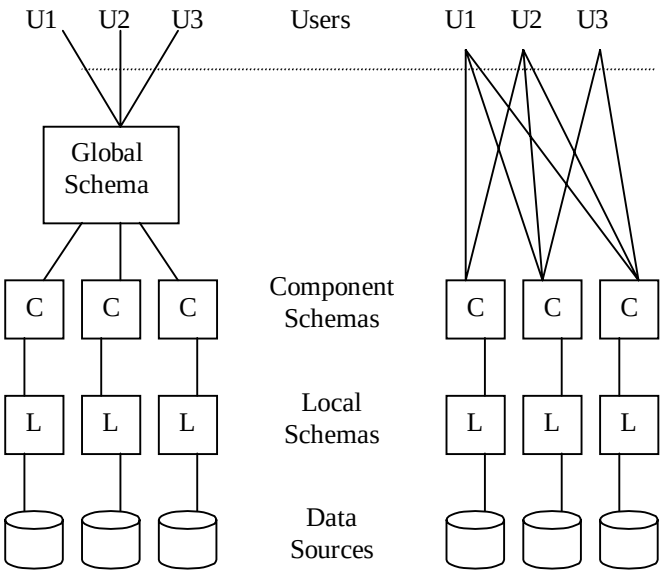

Fig 2.4: Schema architctures for the multidatabase systems discussed in this section. (1) Global schema, (2) Multidatabase language.

A local schema is the conceptual schema of a component database system. For each local schema, there is a corresponding component schema. The component schema represent the same information as the local schema, but the common data model is used instead of the data model of the component database system. A query against a component schema is translated to queries against the underlying local schema. The results of these queries are then processed to form an answer to the initial query.
An integrated or global schema is an integration of multiple component schemas. It makes it possible to access data from multiple databases as if it was stored in a single database.

2.3.3 The global schema approach

One approach to design of a multidatabase system is the idea of a *global schema*. This schema, used by the front end system, is the result of schema integration of the schemas exported from the underlying databases (i.e. the local databases). The export schemas are then in the global data model used by the front-end system. Each export schema is obtained by a schema transformation from the schema of a local database.
The global schema approach can be defined as a single view over the local databases.

However, the creation of a global schema is usually a difficult task. The main reason for this is the lack of a general solution for the semantic conflicts between local schemas in a situation in which everything must be integrated.
Furthermore, even for organizational reasons alone, a single schema for a

large number of component databases is in practice almost impossible to design.
Finally, it is desirable that the individual databases maintain a certain degree of autonomy over their data in order to continue to provide unaltered services for their existing applications and to support controlled access to their information.

2.3.4 The multidatabase language approach

An alternative approach for extracting data from multiple databases is to create a temporary view pertinent to a query.
The users know that they are working against multiple databases.
The multidatabase systems provides a *multidatabase language* powerful enough to support interdatabase queries. Furthermore the multidatabase language must provide a homogeneous interface to its users to hide the differences between the databases by means of multidatabase views.
Users may then themselves select the sources they are interested in and retrieve and combine information from those. It is the user's responsibility to create and maintain the view.

The approach used in our work provide an extension of the AMOSQL query language that offers functions for querying multiple databases and allow users to create temporary views.
A multidatabase language must allow users to manipulate a collection of autonomous databases. Such a language needs features not found in the classical database languages.
Some of the key features required of a language for interoperability in a relational multi database system are the following.

1. The language must have an expressive power that is independent the schema with which a database is structured.
2. The language must be easy to use and sufficiently expressive.
3. The language must provide the full data manipulation and view definition capabilities, and must be compatible with SQL syntax and semantics. This requirement is imposed in view of the importance and popularity of SQL in the database world.
4. Finally, the language must admit effective and efficient implementation.

A significant research problem is the nature of such a language. The component DBMSs may implement special operators in their query languages which makes it difficult to define a language that is as powerful as the component ones. It is also possible that component DBMSs may provide different interfaces to general purpose programming languages. One solution is to implement extensible languages which can be extended by operators for the user defined types. This is why the O-O model is good for data integration.

2.3.5 Commercial standards

In the commercial world middleware products, like ODBC, are often used to solve relational DBMS interoperability problems. To enable multiple RDBMS and application tools to work together they provide an API which gives programmers a uniform SQL interface for accessing different databases.
In this way programmers do not have to handle different SQL dialects, APIs and network protocols. However the products give very little help with integration problems and there is no real multidatabase language. In fact, queries are not allowed to span multiple databases (no joins between tables form different databases).
Another solution that has been used, not very successfully, is the so called gateways for specific pairs of DBMS . For example, gateways are used in the INGRES line of database products and in the ORACLE relational database system. A gateway between system A and system B translates a query in A's language into a equivalent query in B's language and submits the translated query to system B. This solution has a serious limitation. Because it is only concerned with the problem of translating a query expressed in one language into an equivalent expression in another language, rather than with data integration. As such, it does not address the issues of homogenizing the structural and representational differences between different schemas.
After this overview should be clear that the capabilities for multidatabase manipulations present in commercial systems are only a limited subset of those proposed by research.

2.3.6 Object Oriented views

OO views [14] are used to resolve the problem of the access the data in their diverse native formats, (i.e. their semantic heterogeneity), using a common, O-O, data model.
OO views with this purpose are defined over the participating databases to provide the user with a unified appearance of data. To the user the data appears as if it is stored in an OO database.
The queries over the view are transformed to queries over the relational database. The result of these queries are then processed to form an answer to the initial query.
Furthermore OO view can also store its own data and methods. Therefore it must be possible to process queries that combine local data residing in the object view with data retrieved from the database.

This survey of multidatabase system concepts and the role of the common data model to handle the heterogeneity of the participating databases shows that object views of different kinds of data sources is a central issue in multidatabase systems.

2.4 The AMOS O-O multi-database system

This section gives an overview of AMOS which is the framework for the work described in this report. First the multidatabase aspect of AMOS is described. Then we will present its data model and the query and data manipulation language AMOSQL.

2.4.1 The AMOS multidatabase system architecture

The core of the AMOS system is an open light-weight and extensible DBMS. The aim of the AMOS architecture is to provide for efficient integration of data stored in different repositories.
To achieve better performance and because most of the data reside in the data repositories, AMOS is designed as a main-memory DBMS.
The basic way to access data in AMOS is through a multidatabase language. The multidatabase language provide a very flexible way to work with data from multiple data sources. Any combination of data can be used together at any time without the need to define an integrated view of data in advance.

In the wrapper/mediator architecture on which AMOS is based, an intermediate layer of software, called mediator [11], is introduced between the data sources and the applications using them (Appl. 1, 2 and 3 in Figure 2.4.). The applications access the data sources through one or several mediator systems which present high-level abstractions (views) of combinations of data sources. This intermediate layer also increases data independence. It will be possible to make changes in the database layer without having to change the applications.

The software components of the AMOS multidatabase system architecture are shown in Fig. 2.5.

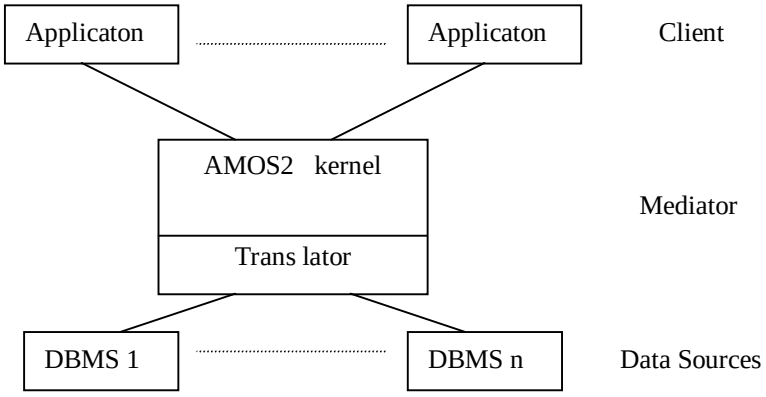

Figure 2.5.

Translators implement the mapping between local schemas (expressed in the data models of the component databases) and the corresponding component schemas (expressed in the common data model). There is one kind of Translator for each kind of data source. A query sent to a Translator is transformed to calls to the underlying data source. The results of these calls are then processed to form an answer to the initial query.

If we want to provide access to some external data source we link its access API with an Amos server (the Translator). Queries can be put directly against Translators using the AMOSQL query language extended with functions providing access to the data sources. These functions will return the information as literal valued information, i.e. as integers or strings.
Then, if we want to combine information retrieved from the data sources, AMOSQL provides multidatabase query features, that allow us to do this.
AMOS provides an O-O model as the framework for uniform interoperation of multiple data sources.

2.4.2 The AMOS data model

The AMOS data model has three basic constructs: objects, types and functions.
Objects are used to model entities in the domain of interest. An object can be classified into one or more types which makes the object an instance of that type. The set of all instances of a type is called the extent of the type.
The properties of the objects and their relationships are modeled by functions. Everything in the data model is an object, including the types and the functions. Each type is represented by an object of the type 'type' and each function corresponds accordingly to an object of type 'function'.

There are two types of objects in AMOS. The literal objects, such as integer, real and charstring, are self identifying. Surrogate objects that have unique, system generated object identifier(OID).

Types are used to describe object structure, they are organized in a OO type hierarchy of subtypes and supertypes. The most general type is the type *object*, all other types are subtypes of object. User defined types are subtypes of a special type called *usertypeobject*.
Functions are defined on types, and are used to represent properties of objects and relationship between objects. Function have an extent, which is the relationship between arguments and results of a function. There exist three different kinds of functions in AMOS; stored, derived, and foreign functions. A stored function has its extent explicitly stored in the database. A derived function uses the AMOSQL query language to calculate the extent. A foreign function is implemented in a general programming language, such as Lisp, C or Java.

Each function has a name and a signature. The latter is the name of the function together with its argument types and result types.
AMOS allow functions to be *overloaded* : the same name can be given to functions defined on different types.
When an overloaded function name is used, the right function is chosen by the system by looking at the types of its arguments and results. This is called function name resolution and the chosen function is called resilient.

2.4.3 The AMOS query language

Data definition and data manipulation in AMOS are performed with the language AMOSQL which is an extension of OSQL.
It is a functional and O-O query language supporting O-O abstraction and declarative queries.
Similar to the SQL's select clause, the *select* clause in Amosql contains a list of variable names or function calls.
The *from* clause contains the names and types of variables used in the select and the where clauses.
The *where* clause contains a predicate expression that may contain functions, variables, constants or nested queries

A simple personnel database can be established by issuing the following AMOSQL statements, which create two new types, their associated attributes and two instances of each type:

```
create type department(Deptno integer, Location charstring);

create function deptno(department dp)-> integer num;
create function location(department dp)-> charstring loc;
```

```
create type employee subtype of person;

create function name(employee em)->charstring nam;
create function ageof(employee em)->integer age;
create function dept(employee em)->department dep;
create function salary(employee em) ->integer sal;


create department (deptno, location) instances
        :sale_dept(11,'S.F.'),
        :support_dept(22,'L.A.');

create employee(name, ageof, dept,salary) instances
        :john('John Smith',53, :sale_dept, 40000),
        :bob('Bob Hanson', 28, :support_dept, 20000);
```

To retrieve the information of all employees located in San Francisco, the request can be formulated in an AMOSQL query as follow:

```
select name(e), ageof(e),deptno(d), salary(e)
from employee e, department d
where dept(e) =d and location(d) = 'S.F.';

<"John Smith",53,11,40000>
```

2.5 ODBC

One way to provide interoperability is with an intermediate software layer, called middleware, that encodes requests and responses into a uniform code. Middleware products have to be sufficiently generic in order to allow a wide range of applications to interact with various data sources.

ODBC [10] is a middleware product, it addresses the multi database connectivity problem using the common interface approach, which gives programmers a uniform interface to different relational databases. The library of functions defined in the ODBC interface allows an application to access DBMSs using the SQL query language.
When using ODBC , an application makes database calls in ODBC format. Dynamically linked drivers- one for every DBMS to be accessed - translate the ODBC calls into calls appropriate to the selected DBMS engine. Requests processed by the DBMS are returned to the application. In this way application developers avoid writing customized code for each application/ DBMS engine connection.
ODBC does not require expensive and time consuming movement of all corporate data into a single data store. It is based on open industry specifications.

3 Design

This chapter describes the approaches used later in the implementation, followed by a description of the architecture and the three main design phases of the system.

3.1 Approaches

The philosophy used to implement our system was to provide a practical solution to the problem of interoperability among a number of component relational databases storing semantically similar information in structurally dissimilar ways.
The requirements for interoperability even in this case fall beyond the capabilities of conventional language like SQL.
The ODBC standard was selected as tool to access SQL based relational DBMSs because it permits to execute SQL calls without customizing the application for DBMS-specific API code and because it has a broad industry support.

Pure multidatabase manipulation languages, like MSQL [12], are not based on the O-O technology. Such languages, extend the traditional functions of SQL to the context of a multidatabase system and are capable of expressing queries over multiple databases in a single statement.
In contrast, our proposal extends the AMOSQL query language with functions that allow SQL access to relational databases. In this work, the basic mechanism is to query single data sources and obtain the result in the mediator. For multi-database views and queries the user has available a general view mechanism in AMOS that provides a great facility for interoperability and data/meta-data management in relational multidatabase systems through a uniform interface. The view facility of AMOSQL can be used to resolve inconsistencies among data from local databases in the multidatabase system by exposing only the appropriate data through the view. This is similar to the approach taken in multidatabase systems utilizing an (integrated) global schema. Our architecture is more flexible and does not require a global schema, yet, the view facility can mimic the role played by the global schema for resolving data inconsistency.
The results in this paper are based on experiences from the

development of views mechanism for the relational databases MSAccess [17] and Interbase [16].

3.2 Architecture

This section describes the architecture of the system component for multidatabase querying.
A highlight of our architecture is that it builds on the existing systems in a non-intrusive way, requiring minimal extensions to the existing database system. This makes it possible to build our system on top of any of the already available SQL systems.

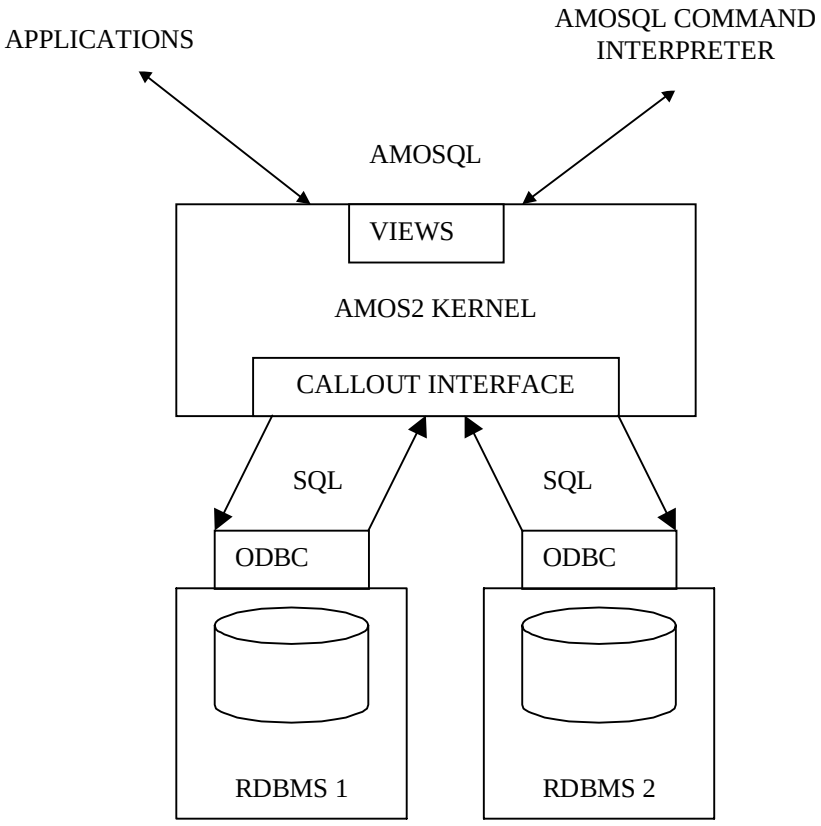

Figure 3.1

Fig 3.1 show the architectural schema of the system. The architecture consists of an Amos II server that communicates with the local databases through the Amos callout interface**.** Such interface allow the Amos II kernel to call external subroutines written in C. We assume that the meta-information comprising databases are stored in the Amos database.
In our architecture, AMOSQL queries are submitted to the Amos II server, which determines a series of local SQL queries(in the form of calls to foreign functions) and submits them to the local databases. The Amos II server then collects and combines the answers from the local databases to produce the answer to the initial AMOSQL query.

3.3 Phases

The primary purpose of this work is to develop an ODBC callout interface for Amos II to allow SQL access to relational databases from AMOSQL through ODBC using Amos II extensibility facilities. The extensibility is supported through foreign functions developed in an external programming language.
Figure 3.2 shows the design phases in our approach.

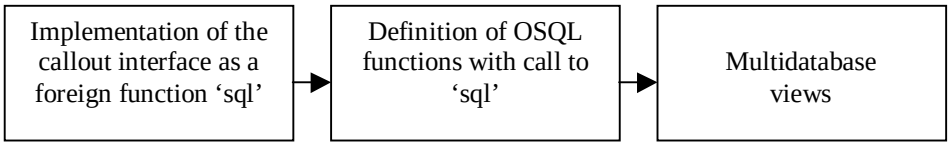

Figure 3.2

The main task in the first step is to develop the callout interface as an Amos foreign function written in C, called 'sql', with which an arbitrary SQL statement over the relational databases can be called.
Then we demonstrate how relational database servers (Interbase and MSAccess) can be called through ODBC from AMOSQL queries. For this purpose we define OSQL functions that call the foreign function.
The second task of this work is to show that different relational databases can be accessed at the same time from Amos II and the results can be combined through a multidatabase view. This is verified by defining OSQL functions that span two different relational databases.

4 Extending AMOSQL with ODBC access

In this section we present the databases we use to illustrate our interface. Then we describe how to use ODBC data sources in Amos. Finally, the design phases presented in the previous chapter are here discussed in a more in deep manner.

4.1 Example multidatabase

The example schemas that follow define the component databases that we use in all examples throughout this chapter. Then we provide a general classification of schematic conflicts that may arise in relational multidatabase systems.

4.1.1 Scenario

Let us assume that a company whishes to combine data modeling employees stored in two different databases. The databases have the schema definition show in Fig 4.1.
The tag name EMPDB identify a database developed with Interbase, DB_EMP identify a MSAccess database.

The databases EMPDB and DB_EMP model employees of the company, furthermore EMPDB describe the departments.
In the real world, the reason why these companies use different DBMSs relative to the same universe may be due to offices autonomy. Because of a reorganization, the company need to combine information across the different offices.

Database EMPDB

EMPLOYEE Table:	
Emp_no*	Integer
First_name	Char(15)
Last_name	Char(15)
Hire_date	Date
Job_grade	Integer
Salary	Integer
Full_name	Char(30)
Dept_no	Char(2)

DEPARTMENT Table:	
Dept_no*	Char(2)
Department	Char(30)
Mngr_no	Integer

Database DB_EMP

EMP Table	
Emp_no*	Integer
Name	Char(30)
Hire_date	Date
Salary	Integer
Job_grade	Char(2)
Dnum	Integer

Figure 4.1. (The character * identifies key attributes).

4.1.2 Schema and data conflicts

All the data in Fig 4.1. is modeled in the relational data model. However, it is semantically heterogeneous. This is due to the databases autonomy and because of the following properties:

1. The databases disagree upon the choice of attributes that should model the universe of employees and the names modeling the same concepts.
2. Types or domains of semantically equivalent attributes may be different.
3. The databases use separate codes to denote distinct grades. For example the job_grade in EMPDB can take the values 1,2,3,4,5 but A,B,C,D,E in DB_ EMP.
4. Some employees may appear in both EMPDB and DB_ EMP, but not all employees are stored in both the databases.
5. Despite the same names, primary key values modeling the same object in different databases are independent.
6. The databases may disagree on the values of vary attributes.

7. In contrast, the databases always agree on a employee name and the corresponding number.

Similar properties will be typical of the general multidatabase environment. Multiple databases relative to the same universe will usually resemble each other, but will also present numerous semantic differences like those mentioned above.

4.2 ODBC data sources in AMOS

ODBC provides an abstraction for the data source that encapsulate all the information (server, database name, user name, password) for linking a client application with a data source.
The ODBC Administrator application allow users to define and use data sources, it associated each data source name with an ODBC driver and all of the information's necessary for the driver to get the data.

The ODBC data source concept is used in our system to implement an object abstraction of it in the form of an AMOS type named 'datasource' that has two properties: name and database type.
This object abstraction has been achieved by defining AMOSQL functions and procedures that calls the more low level primitives (foreign functions) we have built, namely:

- the foreign function datasources() returns the ODBC data sources are available in the system,
- name(datasource) and type(datasource) retrieve the name and type of the data sources, respectively.

The meta-information about the data sources is stored in the AMOS database. So, the user can connect to the AMOS database and access the meta-data without the data source being connected through ODBC connection functions.
Users may then themselves select the sources they are interested in and retrieve and combine information from those.
Furthermore, user can query what meta-data is available in each of the data sources using the following foreign functions:

- tables(datasource)-> tables names
- columns(datasource, table name) -> <columns names,data types>

4.3 How to define an OSQL function that calls an arbitrary SQL statement

As it was pointed out in section 3.3, the first task to build our system for implementing a multidatabase query facility is to allow SQL access to the relational databases from AMOSQL.
To implement this task a callout interface for Amos II has been developed as a foreign function written in C, using the external interface between Amos II and the C programming language. The implementation of the foreign function is described in chapter 5.
The foreign function, called 'sql', has the following signature:

```
sql(charstring data source, charstring query, vector parameters) -> vector results
```

The function takes 3 parameters. The first is the name of the ODBC data source that we want to query with the sql function. The parameter named query is the SQL statement with which we want to retrieve data from the database. The SQL statement is often in a parameterized form, in which case the parameters of the statement are passed through the last parameter of the sql function, the vector named parameter.
Parameters are markers within an SQL statement that represent where a value goes. The actual value of the parameter is specified at execution time.
Parameter help to create extremely interoperable applications.

Once the SQL function has been defined, the data in the relational databases can be selectively made accessible from Amos II by defining an OSQL function that call an arbitrary SQL statement through the 'sql' function.
For example, the following function retrieves from EMPDB all the names of the employees working in a specified department.

Example 1, function for querying a single database:

```
create function empname(charstring  dept)->integer ename as select ename where
sql("EMPDB","SELECT FULL_NAME FROM
EMPLOYEE E, DEPARTMENT D WHERE E.DEPT_NO=D.DEPT_NO
AND D.DEPARTMENT=?",{dept})={ename};
```

The "EMPDB" data source is the ODBC abstraction for the Interbase relational database described in section 4.1.

4.4 How to use the SQL function for multi-database views.

Multidatabase views capabilities to support data independence are provided by advanced O-O query languages. In AMOSQL, views are supported through derived functions.

In the previous section we have shown how to access relational databases from Amos II with a derived OSQL function that uses the AMOSQL query language and a call to the 'sql' function to calculate the extension.
Here we present how different relational databases can be accessed at the same time from Amos II and the results can be combined through a multidatabase view.
In order to allow multidatabase views it is necessary to define derived AMOSQL functions that have multiple calls to the sql function for different relational databases.

It is, so, possible to combine information from different sources, i.e. perform multidatabase joins. For example, using the databases described in section 4.1 we can retrieve all the departments in 'EMPDB' whose manager, stored in 'DB_EMP', is specified in the argument of the following query:

Example 2, function for multidatabase join:

```
create function multijoin(charstring name)-> charstring dept
as select dept  from integer num where
sql("DB_EMP","select emp_no from emp where
name=?",{name})={num}
and sql("EMPDB","SELECT department FROM department WHERE
mngr_NO = ?",{num})={dept};
```

Inconsistency of data among data sources are common in multidatabase systems, our architecture provide means to cope with these difficulties.
For example, the company may need to retrieve the salary of employees stored in both databases to check for inconsistency. For this purpose we define a function that retrieves the salary and name of the employees belonging to both the databases :

Example 3, function for querying multiple data sources:

```
create function amount()->< integer am1, integer am2, charstring
name>
as select am1, am2, name where
sql("EMPDB","select salary, full_name from employee",{})={am1,name}
and sql("DB_EMP","select salary ,name from emp",{})={am2,name};
```

As we can see in the previous example, the user needs to formulate as many SQL queries as there are databases, furthermore these queries may differ from database to database, this is due to the fact that a given concept may be expressed in different ways.

Another example of how to use a derived function to broadcast the same manipulation to several databases (e.g. project any relation describing employees on the attribute expressing the hire date) is the following

function that retrieve the name and hire date of all the employees stored in the two databases excluded copies, having the maximum jog grade.

Example 4, function for querying multiple data sources:

```
create function maxim() -><charstring name, charstring hdate>
as select name, hdate  where sql("EMPDB","select full_name, hire_date
from employee where job_grade =’5’",{})={name, hdate}
or  notany ((select name from charstring name where
sql("EMPDB","selectfull_name from employee",{}) = {name}))
and sql("Db_Emp","select name, hire_date from emp where job_grade
= ‘A’",{})={name, hdate};
```

5 Implementation

In this chapter some implementation specific details will be discussed, starting with the description of the ODBC tools and techniques. Is then presented how the Amos's streamed callin-callout interface is used in our system with ODBC facilities.

5.1 ODBC

The ODBC interface allows applications to access data from a wide range of DBMSs. Each application uses the same code to communicate with many data sources through DBMS-specific drivers. Therefore, these applications are independent of a particular DBMS. A Driver Manager is located between the applications and the drivers, providing a mutual understanding of the exchanged messages and a common interface for all applications.
In this way, an application developer can write an application for a virtual database and allow a loadable driver to map the logic to the specific DBMS used by the application.
The ODBC advantages come from the portability and interoperability of an application's code.

5.1.1 The ODBC architecture

The *ODBC architecture* has four components:

· *Application*. Performs processing and calls ODBC functions to submit SQL statements and retrieve results.
· *Driver Manager*. Loads and unloads drivers on the behalf of an application. Processes ODBC function calls or passes them to a driver.

· *Driver*. Processes ODBC function calls, submits SQL requests to a specific data source, and returns results to the application. If necessary, the driver modifies an application's request so that the request conforms to syntax supported by the associated DBMS.
· *Data source*. Consists of the data the user wants to access and its associated operating system, DBMS, and network platform (if any) used to access the DBMS.

The following figure shows the relationship between these four components.

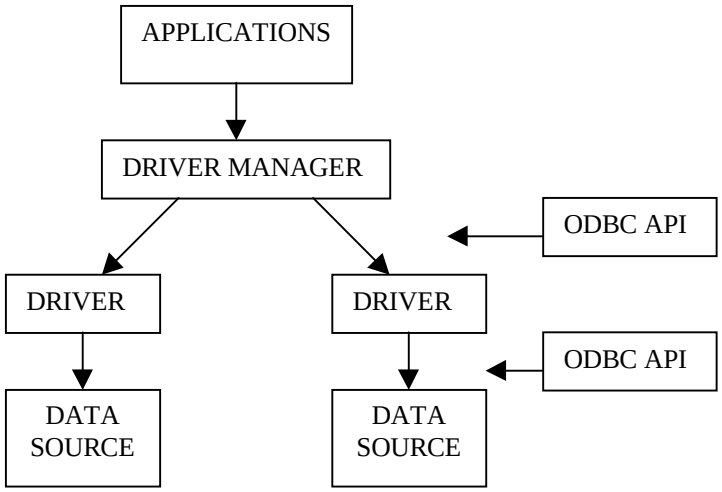

Figure 5.1 ODBC ARCHITECTURE

While the primary purpose of the ODBC Driver Manager is to load drivers that implement the ODBC function calls, the *drivers* themselves implement ODBC function calls and interact with a data source. The driver manages the communication protocols between the application and the data source when the application makes a call to connect to the data source. With a connection to a data source established, a driver is capable of handling requests to the DBMS that are made by the application, providing the necessary data translations to and from the data source, and returning results to the application.

5.1.2 The ODBC interface

The interface functions are grouped as follow:

- Allocate and deallocate:
  This set of functions can allocate the necessary handles to define a database environment, a database connection and a single SQL statement. Executing an allocate function will allocate the needed memory and define the data structures used on the implementation needs and assigns a value to the handle that refers to these data structures. Once it has been allocated, the application can pass the handle to subsequent interface functions to indicate the environment, connection, or statement on which the function should act.
  Similarly, the deallocate functions free the various handles and the memory associated with each.
- Connection:
  Once the handles have been allocated, a connect handle or handles can be established, similarly with statement handles.
- Executing SQL statements
  There are two ways to specify and execute SQL statements: prepare and direct. A prepared execution is used when the application submits the SQL statement multiple times, possibly with changes to the parameter values. The other method, direct execution , is used when the application does not require information about a result set prior to completing the SQL request and we plan to have the application submit the SQL request only once.
- Receiving results:
  This set of functions handles retrieving data and metadata from an SQL statement's result set. As, for example, describing a column of a result set and its attributes, getting the next row of a result set, counting the number of rows that are affected by an SQL statement, and so on.
- Transaction control
- Error handling and miscellaneous
  This function returns the error information that is associated with a handle. The other function in this group allows you to attempt to cancel a SQL statement.

Also, ODBC defines not only a common set of functions to access different databases, but also a common SQL grammar. The SQL language forms one of the key elements in creating interoperable ODBC applications. The API defines only an interface between an application and a database, it does not have functions to open tables or to retrieve rows from a table. SQL does this work.

The ODBC API functions are divided into **three standard**

conformance level called respectively Core, Level1 and Level2. The first one is strictly an implementation of the CLI developed by the SAG which can be seen as the lowest -common-denominator.
The second puts forward new functions for gaining information about the driver and the data source and for setting and retrieving the driver options. The last allows ODBC to manage sophisticated databases functions. An ODBC driver has to implement at least the functions in the Core level and process any part of the SQL grammar defined by ODBC.
An application accesses a data source by function calls of one or more levels and submits SQL requests depending on the driver conformance. Application developers must decide whether to use the minimum level of functionality, or write the conditional code to test for extended functionalities. In either case, the Core level guarantees a functional application.

ODBC also permits developers to use DBMS-specific functions that are not part of ODBC.
Support for proprietary features is important. No matter how far ODBC evolves, there will still invariably be DBMS vendors providing features outside the specification .
The drawback , of course, is that the code becomes less portable.

5.2 Using streamed callin-callout interface in Amos II through ODBC

This section show how our system is interfaced with Amos II and how is used ODBC technology in the task as tool to access different relational databases.

5.2.1 Principles

There are external interfaces between Amos II and the programming languages C, LISP and JAVA. The external C interfaces are intended to be used by system developers that need access to the Amos II system. It is thus implemented on a rather high level and does not directly give access to Amos II internal primitives.
There are two ways to interface Amos II with other programs : either an external program calls Amos II through the callin interface, or Amos II calls external subroutines through the callout interface.
Essentially, to implement foreign AMOSQL functions through the callout interface a number of functions written in the programming language C are needed.

We focus the attention on the implementation of the **'sql'** foreign function, the other foreign functions used in our system, namely: **columns**, **tables**,

datasources, **dbname** and **dbtype**, have been developed using the same concepts.

5.2.2 The driver program

In order to use the callout interface a driver program is needed. It is a C function (the main program) that sets up the Amos II and ODBC environment and then calls the Amos II system.
The driver program contain the following code to define our foreign functions:

- Amos system functions, ODBC API functions and C code to implement the function (described in the next section).
- A binding of the C entry point to a symbol in Amos II
- A definition of the foreign AMOSQL function.
- A call to the ODBC function SQLAllocEnv to allocate an environment handle. ODBC uses this handle to keep track of the database connections that the application has established. It is necessary to establish only one environment in the application because any number of connections can be made within a single environment. The driver will free the environment handle by calling SQLFreeEnv.

The complete driver program with some declarations of Amos II handles look like this:

```
  #include "callout.h"
  HENV henv;    /* Declare a variable of type HENV(the application's
environment handle) */

  main(int argc,char **argv)
  {
/* declares a C handle to hold connections to Amos II */
  dcl_connection(c);
/* fn is a handle to the Amos II function 'sql'*/
  dcl_oid(fn);
/* To hold result streams from Amos II  function calls */
  dcl_scan(s);
```

```
/* Control gets here in case of fatal Amos II errors */
  if(setjmp(resetlabel))
  {
  printf("Fatal error...\n");
  exit(1);
  }
/* Pass the address of henv(the application's environment handle) to
SQLAllocEnv. SQLAllocEnv allocates memory for an environment handle and
initializes the ODBC call-level interface for use by an application. An application
must call SQLAllocEnv prior to calling any other ODBC function.*/

  SQLAllocEnv(&henv);

  init_amos(argc,argv); /* Initialize embedded Amos */
  a_connect(c,"",FALSE); /*Connect to embedded Amos*/


/*Bind C function 'SQLbbbf' to Amos symbol named 'SQLbbbf'*/
  a_extfunction("SQLbbbf",SQLbbbf);

/* Define AMOSQL function 'SQL(a,b,c)->x' to be defined as SQLbbbf when a
and b and c are known while x is unknown: */
  a_execute(c,s,"create function SQL(charstring a, charstring b, vector c) ->
  vector x as foreign 'SQLbbbf';", FALSE);
  a_setf(fn,a_getfunction(c,"charstring.charstring.vector.SQL->vector",FALSE));

/* free resources held by handles */
  free_oid(fn);
  free_scan(s);
  free_connection(c);
/*The driver call the interctive AMOSQL top loop from C by*/
  amos_toploop("Amos");
/* Application finish free the environment*/
  SQLFreeEnv(henv);
return 0;
}
```

5.2.3 Function implementation

Many of the concepts supported by the ODBC API are common to CLI SQL programming. The basic implementation steps are shown in Figure 5.2.

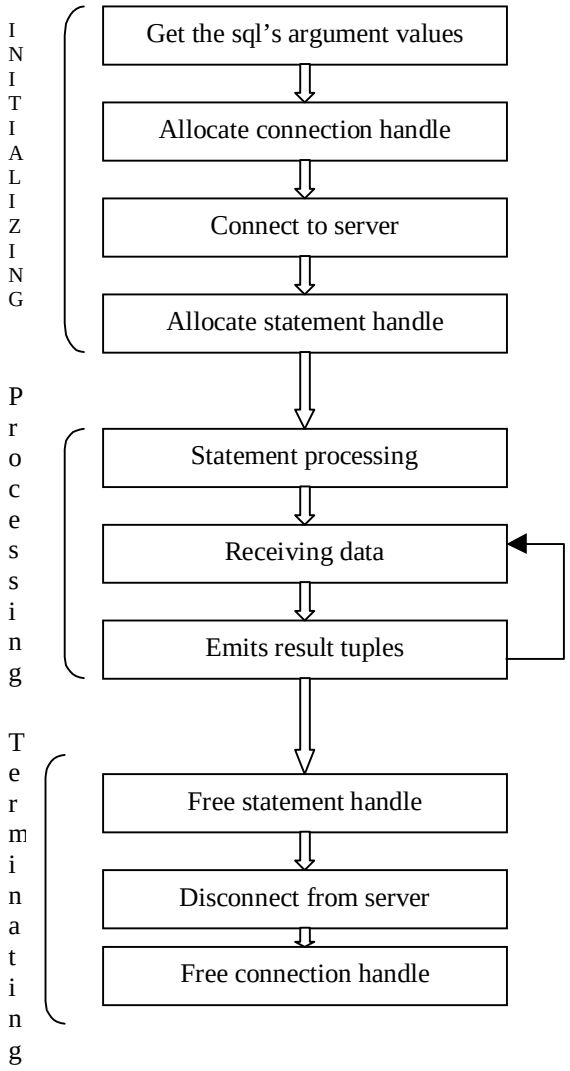

Figure 5.2 Basic flow control for ODBC application

The 'sql' foreign function implementation in C has the signature:

```
void sqlbbbf(a_callcontext cxt, a_tuple tpl);
```

where cxt is an internal Amos II data structure for managing the call and tpl is a tuple containing the argument values followed by the unassigned result values of the function.
The size of the tuple tpl is the sum of the number of arguments of the AMOSQL function (its arity) and the width of its result tuple. The 'sql' function has three arguments and one result, so the size of tpl is 4.
Tuples are used for interfacing foreign Amos II functions defined in C when passing data from Amos II to C and vice versa. Furthermore, tuples are used for representing sequences.

In the implementation of the 'sql' function, tuples are used to get the argument values first. There are a number of Amos II system functions for moving data from the tuples to C data areas and to copy data from C data areas into tuples.
For example to get the first argument of the function, representing the data source name, into a C string we have used the system function :

```
a_getstringelem(tpl, 0, DSName, sizeof(DSName), FALSE)
```

It copies the element in position number 0 of the tuple tpl, that is a string, into the C variable DSName.
The same function is used to get the value of the argument representing the SQL query. Data values of type vector, like the third argument of the function (used to pass the parameters of the SQL query), are represented as sequences. Sequences are special Amos II objects that hold references to several other objects. Each object handle has an associated data type that specifies what kind of object the handle references.
To get the values of the parameters of the query (if any), the sequence stored in position number 2 of the tuple tpl is converted into a tuple argl with the following function:

```
a_getseqelem(tpl, 2, argl, FALSE);
```

Then each element of the tuple argl is gotten using the functions:

```
a_getstringelem(argl,i,stringa,sizeof(stringa),FALSE);
a_getintelem(argl, i, FALSE);
a_getdoubleelem(argl, i, FALSE);
```

if the element of position i in the tuple argl is of type string, integer, or real rispectively.

ODBC uses handles to track the storage and context also for connections and statements. The application then allocates a handle for a database connection using SQLAllocConnect before calling the connection function SQLConnect. SQLConnect loads a database driver and establishes a connection to the data source specified.
As part of its termination logic, the application will close and free the connection handles by calling SQLDisconnect and SQLFreeConnect.

Next, each query's parameter is bound to its values with the SQLBindParam command. Binding a parameter means tying it to a storage location from which the parameter's value can be retrieved at execution time. When the parameters are bound the SQL statement is executed and a result which can contain several data rows is generated.
To execute an SQL statement we use the prepared execution mode which provides great flexibility, especially when using SQL statements with

parameters. The advantage of prepared execution are detailed in the following section. To process statements, the application must first acquire a handle by calling SQLAllocStmt. To free statement handles, the program calls SQLFreeStmt.

Before retrieving data from a result set, the application determine certain information about the set itself using the SQLDescribeCol that returns the result descriptor (name, type, precision, scale, and nullability) for each column in the result set.
Once the application has retrieved information about a result set , it is ready to retrieve the result set itself. There are two functions for retrieving results SQLbindCol and SQLFetch. The first specifies storage for result set data, the latter retrieves the data into a storage location.
SQLFetch returns one row of data at a time. This technique of fetching a bound result set is of central importance in our interface since to retrieve data from results of foreign functions calls the callout interface uses a stream of tuple. For the purpose of return the results of the 'sql' function calls as a stream of tuple, each fetched row is converted as a tuple (called resseq in the following examples) before a new call to SQLFetch is executed.
Next the 'sql' function emits result tuples of the same size as tpl with the result variable positions filled in. The tuple tpl is therefore updated as in the following examples where the result position 3 is set to the result sequence values:

```
a_setseqelem(tpl, 3, resseq, FALSE);
```

Finally, the result tuple is returned to Amos II through the function :

```
a_emit(cxt, tpl, FALSE);
```

ODBC functions can post zero, one, or more errors, so checking for errors after calling SQLError is good programming practice (defensive programming). The application call SQLError when ODBC functions return SQL_ERROR or SQL_SUCCESS_WITH_INFO. To get a clear representation of status, we check SQLState and make repetitive calls to SQLError until ODBC returns SQL_NO_DATA_FOUND.

5.3 Prepare optimization

Before discussing the techniques for using the prepare execution, it is necessary to discuss how an SQL statement is processed.
To process an SQL statement, a DBMS performs the following five steps:

1 The DBMS first parses the SQL statement. It breaks the statement up into individual words, called tokens, makes sure that the statement has a

valid syntax, and so on. Syntax errors and misspellings are detected in this step.

2 The DBMS validates the statement. It checks the statement against the system catalog. Check if the tables named in the statement exist in the database, and if they contain the used columns. The user privileges are also checked at this stage. Certain semantic errors can be also detected in this step.
3 The DBMS generates an access plan for the statement. The access plan is a binary representation of the steps that are required to carry out the statement; it is the DBMS equivalent of executable code.
4 The DBMS optimizes the access plan. It explores various ways to carry out the access plan. Can an index be used to speed a search? Should the DBMS first apply a search condition to Table A and then join it to Table B, or should it begin with the join and use the search condition afterward? Can a sequential search through a table be avoided or reduced to a subset of the table? After exploring the alternatives, the DBMS chooses one of them.
5 The DBMS executes the statement by running the selected access plan.

The steps used to process an SQL statement vary in the amount of database access they require and the amount of time they take. Parsing an SQL statement does not require access to the database and typically can be done very quickly. Optimization, on the other hand, is a very CPU-intensive process and requires access to the system catalog. For a complex, multitable query, the optimizer may explore thousands of different ways of carrying out the same query. However, the cost of executing the query inefficiently is usually so high that the time spent in optimization is more than regained in increased query execution speed. This is even more significant if the same optimized access plan can be used over and over to perform repetitive queries.

Prepared execution is an efficient way to execute a statement more than once. The statement is first compiled, or prepared, into an access plan by calling SQLPrepare. The access plan can be then executed one or more times at a later time with SQLExecute. The prepared execution is faster than the direct execution for statements executed multiple times, primarily because the statement is compiled only once; statements executed directly are compiled each time they are executed.
Prepared execution can also provide a reduction in network traffic because the driver sends an access plan identifier to the data source each time the statement is executed, rather than the entire SQL statement. While we might not call the exact same SQL statement multiple times, the same prepared statement, with the use of parameters, can act like a number of different statements at execution time.

6 Performance Measurements

Measurements have been made on the interface we have developed, which will be presented in this chapter. First the setup for the benchmark is described, then the measurements are presented in diagrams.

6.1 Setup

The tests have been focused on the time to access Interbase and Access both individually and in multi-database views.
The benchmark test use an employee and department table with columns representative of typical data, such as salary and hire date. The test execute queries that retrieve data using SELECT statements. In simple terms, each test performs the same operations against identical data but varies the type of database (Interbase or Access) and the size of the database.

The diagrams shows the execution time for the following queries:

Qry 1 = select by employee number
Qry 2 = join
Qry 3 = multi-database view as multi-database joins

The benchmark summary presented here represents multiple tests using tables populated with 1000, 5000 and 10000 employees and 100 departments. Multiple runs provided execution times used to compute a mean time for each test. The tests include logic that uses various SQL programming techniques, such as direct or immediate execution queries and prepare and execute queries.

6.2 The measurements

The first and second diagrams show the execution time to access Interbase and Access individually, the third diagram shows the time to execute a multidatabase view.

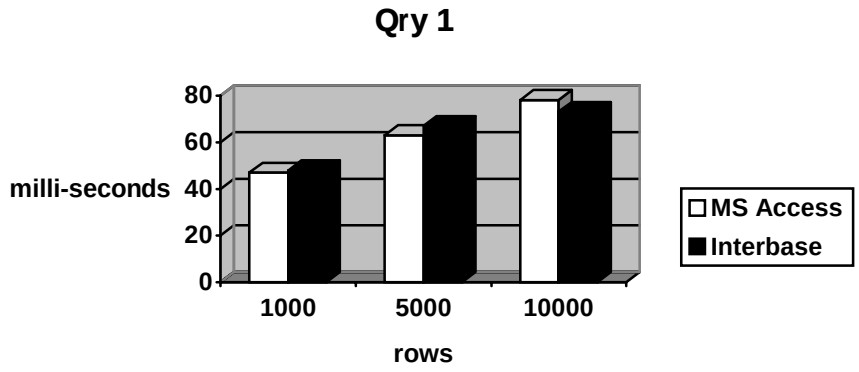

Figure 6.1. First measurement, select by employee number.

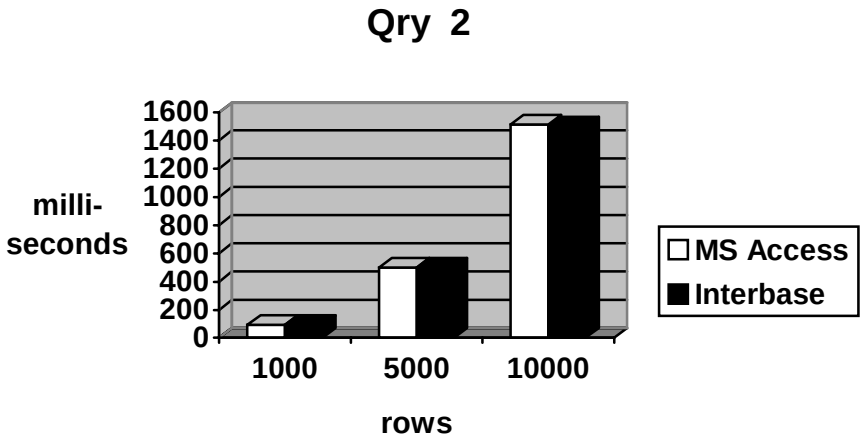

Figure 6.2. Second measurement, multiple join in the employee and department tables.

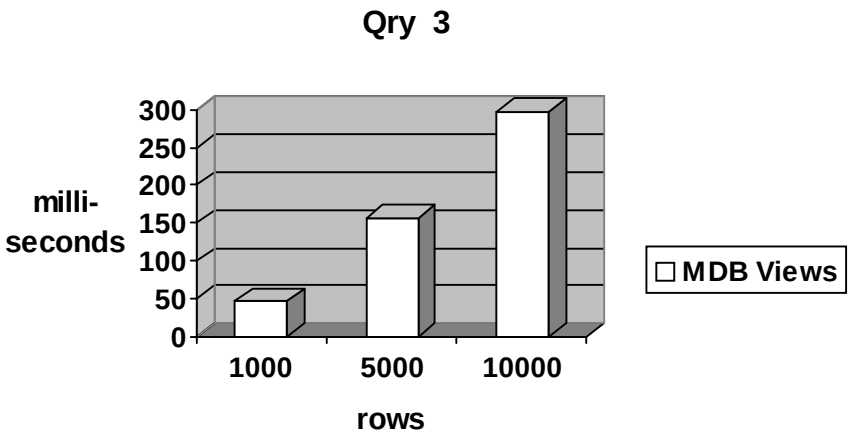

Figure 6.3. Third measurement, multi-database join.

The first and second measurements (Fig. 6.1 , 6.2.) permit to investigate how the data access cost varies for different size of a query to a single relational database. The result is that there are not considerable differences between accessing Interbase versus MSAccess.

The third and final measurements (Fig. 6.3.) show the good performance of the system for multi-database views.

A future work is to include in the tests logic that uses various SQL programming techniques, such as direct or immediate execution queries and prepare and execute queries; and to investigate further preparation cost versus data access cost versus size of a query.

7 Conclusions and Future Work

We have developed an SQL interface for the Object Oriented mediator system Amos II to access multiple, heterogeneous and distributed relational databases using ODBC capabilities.
The interface abstraction has been achieved by the 'sql' function.

Characteristic features of this approach :

- Using the ODBC standard, any relational DBMSs for which a driver is available can be accessed.
- Amos II extensibility is used through the streamed callout interface to access the data sources.
- The solution of the relational DBMSs interoperability problem involves constructing multi-database views in Amos as homogeneous interfaces from the desired set of data sources.
- The evolutionary facility of the object abstraction of the ODBC data source concept, make it possible to change the set of data sources involved in the system.

Our initial experience of building some multi-database views and the performance measurements have demonstrated the feasibility of this approach.
In future our work can be combined with an Object Oriented view mechanism to built an abstraction of the data/meta-data retrieved with our interface. With this abstraction the AMOS mediator is capable of tackling schematic discrepancies among the heterogeneous databases in the system.

APPENDIX

Here we show a complete run example of the interface developed in our work, including how to connect data sources from Amos II and how to retrieve information about the data sources. The interface is run from the Dos prompt with an argument that is the Amos database where are stored the meta information about the data sources in the form of object abstraction.

```
C:\> sqllast meta.dmp
```

If the set of data sources in the system that we want access through the interface is changed, we then need to update the Amos database. For this purpose we call from the Amos prompt the function **updateds()** that will automatically update the set of Amos data sources fetching all the ODBC data sources involved in the system, and for each Amos data source associates the name and the database type.

```
Amos 1> updateds();
```

The new Amos set of data sources is then stored in the Amos database with the command:

```
Amos 2> save ”meta.dmp”;
```

At this point we can start to query the database previously stored. For example we may need to know the name and type of each data source:

```
Amos 3> select name(d), typeofds(d) from ds d;

<”EMPDB”,”InterBase 4.x Driver by Visigenic (*.gdb)”>
<”LIBDB”;”Microsoft Access Driver (*.mdb)”>
<”DB_EMP”,”Microsoft Access Driver (*.mdb)”>
```

Once we know the data sources available in the system, we can start to retrieve information about the data sources themselves. This is achieved by the functions **tables(data source)** and **columns(data source, name of the table)**. The first function retrieves all the tables of a specified data source, the latter retrieves all the columns and the associated data type.

To retrieve meta-data with the functions presented above and to retrieve data with the 'sql' function described in chapter 4 and 5, the data sources must be connected. For this purpose is available a function called **odbcsource(data source, user name, password)** that explicitly open a connection for a specified data source.
To disconnect data sources the function **closeodbc(data source)** must be called.

```
Amos 4> odbcsource("EMPDB","SYSDBS","masterkey");
Amos 5> odbcsource("DB_EMP");

Amos 6> tables("EMPDB");
"DEPARTMENT"
"EMPLOYEE"

Amos 7> columns("EMPDB","EMPLOYEE");
<"EMP_NO","SQL_SMALLINT">
<"FULL_NAME","SQL_VARCHAR">
<"DEPT_NO","SQL_CHAR">
<"SALARY","SQL_NUMERIC">
```

Follow a description of how to use the 'sql' function to retrieve data, starting with an example of query to the Interbase database. Then an example of multidatabase view is presented.

```
Amos 8> create function empname(charstring  dept)->integer ename as
select ename where sql("EMPDB","SELECT FULL_NAME FROM
EMPLOYEE E, DEPARTMENT D WHERE E.DEPT_NO=D.DEPT_NO
AND D.DEPARTMENT=?",{dept})={ename};

Amos 9>empname("Marketing");
```

```
Amos 10>create function multijoin(charstring name)-> charstring dept
as select dept  from integer num where
sql(”DB_EMP”,”select emp_no from emp where
name=?”,{name})={num}and sql(”EMPDB”,”SELECT department FROM
department WHERE mngr_no = ?”,{num})={dept};

Amos 11> multijoin(”Lee, Terri”);
```

REFERENCES

[1] W.Litwin, T.Risch: 'Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates', IEEE Transaction on Knowledge and Data Engineering, Vol. 4, No. 6, December 1992.

[2] G.Fahl,T.Risch, M.Sköld : 'AMOS - An Architecture for Active Mediators', Proc. Intl. Workshop on Next Generation Information Technologies and Systems, NGITS '93, Haifa, Israel, June 1993.

[3] D.W.Shipman : 'The Functional Data Model and the Data Language DAPLEX',ACM Transactions on Database Systems, Vol.6, No.1,March 1981.

[4] Lyngbaek, P. : 'OSQL: A Language for Object Databases', HPL-DTD-91-4,Hewlett-Packard Company, January 1991.

[5] Kim W.,Lochovsky F. H. : 'Object-Oriented Concepts, Databases and Applications', ACM Press, Addison Wesley,1989

[6] Cattell, R. G. G. : 'The Object Database Standard:ODMG-93', Morgan Kaufmann Publishers, Inc., 1994.

[7] Melton, J. ANSI SQL3 Papers SC21N9463 - SC21N9467, New York, U.S.A. 1995.

[8] R.Elmasri, S.B.Navathe : 'Fundamental of Database Systems', The Benjamin/Cummings Publishing Company, Inc. 1989

[9] A.P. Sheth, J.A. Larson : ' Federated Database Systems for Managing Distributed, Heterogeneous and AutonomousDatabases', ACM Computing Surveys, Vol 22, No.3, September 1990.

[10] Signore, Creamer, Stegman : 'The ODBC Solution', MC Graw Hill, 1995.

[11] Wiederhold, Gio : ' Mediators in the Architecture of Future Information Systems'. IEEE Computer, March 1992.

[12] W. Litwin :'MSQL: a multidatabase language' Information science, 1989

[13] E. Codd :' A Relational Model for Large Shared Data Banks', Communications of the ACM, Vol.13, No. 6, june 1970.

[14] G. Fahl : 'Object View of Relational Data in Multidatabase Systems', Licentiate Thesis No 446, Department of Computer and Information Science, Linköping University, 1994.

[15] V. Josifovski : ' Calculus-Based Transformations of Queries over Object-

Oriented Views in a Database Mediator System'.

[16] Interbase Software Corporation : http://www.interbase.com/

[17] 'Microsoft Access user's manual' , Microsoft Press.