Master's thesis

An ODBC-driver for the mediator database AMOS II

Marcus Eriksson

LiTH-IDA-Ex-99/50

1999-05-06

Linköping University
Department of Computer and Information Science

An ODBC-driver for the mediator database AMOS II

Marcus Eriksson

LiTH-IDA-Ex-99/50

1999-05-06

Supervisor: Professor Tore Risch

Abstract

ODBC (*Open Database Connectivity*) is a standardized application programming interface (*API*) developed by Microsoft. By using the ODBC interface, applications can access a wide variety of data sources using the same source code. Prior to ODBC, applications written to access data stored in a Database Management System (*DBMS*) had to use the proprietary interface specific to that database. If application developers wanted to provide their users with access to data in more than one data source, they needed to code to the interface of each data source. Naturally, applications written in this manner are difficult to code, difficult to maintain, and difficult to extend.

The ODBC architecture was designed to permit maximum interoperability. It allows application developers to create an application without targeting a specific DBMS. End users can then use the application with the DBMS that contains their data by adding modules called database *drivers*, which are dynamic-link libraries (*DLLs*).

Today, ODBC has become the industry standard for interoperability with relational databases. Database management systems with ODBC-drivers can interoperate with hundreds of different applications.

AMOS II is a light-weight, main-memory, object-relational database kernel, running on the Windows NT platform. It contains a relationally complete query-language, AMOSQL.

The purpose of this work is to develop an ODBC-driver for AMOS II.
Since ODBC uses SQL as its query language and AMOS II uses AMOSQL, the driver must translate SQL queries into the corresponding AMOSQL query. Moreover, since ODBC was developed with relational database systems in mind, and AMOS II is an object-oriented system, some method of mapping between the two systems must be used.

Preface

This report is a part of a master's thesis in Computer Science. The main part of the work, however, mainly consists of C-code (about 4500 lines). The work was carried out at EDSLAB (Engineering Databases and Systems Laboratory), one of the research laboratories at the Department of Computer and Information Science (IDA) at Linköping University. The work is a part of the AMOS project, whose purpose is to develop and demonstrate a mediator architecture for supporting information systems where applications and users combine and analyse data from many different data sources.

I would especially like to thank my supervisor Professor Tore Risch at EDSLAB for his invaluable support and almost infinite enthusiasm.
I would also like to thank my family for their support during all my years at the University, Sara for bearing with me during all pointer problems and segmentation faults, and of course all my friends at Linköping University.

Linköping, April 1999

Marcus Eriksson

Contents

1 Introduction

In this chapter the background and objective of the thesis is described, as well as limitations and an overview of the report.

1.1 Background

In recent years the need to effectively process large amounts of data has become increasingly important. Companies are using databases to store information about for example warehouses, orders, invoices and much more. Because of this, the need to access this data using standard tools has become very important. Since there are many different kinds of databases in use today, a standard method for accessing data was developed by Microsoft, *Open Database Connectivity* (ODBC). Applications using ODBC as their data accessing method can read and manipulate databases for which there are ODBC-drivers.
The purpose of the AMOS project is to develop and demonstrate a mediator architecture for supporting information systems where applications and users combine and analyse data from many different data sources. A data source can be a conventional database but also text files, data exchange files, web pages, programs that collect measurements or even programs that perform computations and other services.
Previously, the AMOS system lacked a nice interface to the user. The only way of manipulating the database was to write AMOSQL queries on the command-line. However, a Java interface has been developed (Goovi). By developing an ODBC-driver for AMOS, all ODBC applications can access the AMOS database in exactly the same way as they access other databases, for example Oracle or DB2.

1.2 Objective

The main objective of this thesis is to develop an ODBC driver for AMOS. SQL statements has to be parsed into the corresponding AMOSQL queries. Also, mapping between the relational model of ODBC and the object model of AMOS has to be developed.

1.3 Limitations

The goal of this work is not to produce a full-fledged ODBC-driver. Only a subset of SQL will be supported (simple SELECT queries). As will be shown later, the possibility to use AMOSQL and thereby bypassing the parser still allows the user to manipulate data.

1.4 Report overview

This report consists of 9 chapters and 5 appendices. The first chapter gives a short background and a description of the work while chapter 2-4 gives an overview of database systems in general and a closer look at AMOS and ODBC. Users familiar with database technology could skip chapter 2 and proceed directly to the chapter on the AMOS system (chapter 3).
Chapters 5-6 describes the driver design process and the implementation of the driver. Chapter 7 describes some of the problems encountered during the work and chapter 8 gives a small demonstration of a session using the driver with Microsoft Query, the tool used by the Microsoft Office applications to access ODBC data sources. The chapter mainly consists of screen dumps from Microsoft Query during the session. Chapter 9 discusses the result and some possible further improvements to the driver.
The appendices contains more details, such as the source for stored AMOS functions, grammar description and a description of the example database used in chapter 9.
Some parts of the work are not covered in this report, for example driver installation, set-up and de-installation. This mostly consists of adding and manipulating keys in the Windows Registry, which is not very interesting to read about.

2 Database systems overview

2.1 Relational databases

A database system is essentially nothing more than a computerized record-keeping system. The database itself can be regarded as a kind of electronic filing cabinet; in other words, it is a repository for a collection of computerized data files. The user of the system is given facilities to perform a variety of operations on such files, including the following (among others):

- Adding new, empty files to the database
- Inserting new data into existing files
- Retrieving data from existing files
- Updating data in existing files
- Deleting data from existing files
- Removing existing files from the database

There are two kinds of information in a database, the *data* and a *schema*. The schema is metadata describing the semantics of the data in the database. Each DBMS supports a *data model* (the type of data abstraction used to provide a conceptual representation of data without revealing the details of how the data is stored). The most common data model is the *relational data model,* introduced by Dr. E.F. Codd in 1970.

Let’s begin by defining a *relational database management system* as a system in which, at a minimum:

- The data is perceived by the user as tables (and nothing but tables)
- The operators at the user’s disposal (for data retrieval) are operators that generate new tables from old, and those operators include at least SELECT, PROJECT and JOIN

A sample relational database, the departments-and-employees database is shown in the tables below. As can be seen, the database can be “perceived as tables”.

DEPT#	DNAME	BUDGET
D1	Marketing	10M
D2	Development	12M
D3	Research	5M

Table 2-1. Department table (DEPT).

EMP#	ENAME	SALARY	DEPT#
E1	Jones	40K	D1
E2	Ewing	30K	D1
E3	McCoy	45K	D2
E4	Chekov	25K	D3

Table 2-2. Employee table (EMP).

There are certain rules that the database must obey if it is to conform to the prescriptions of the relational model. To be specific:

- Each row in the DEPT table must include a unique DEPT# value. Likewise, each row in the EMP table must include a unique EMP# value.
- Each DEPT# value in table EMP must exist as a DEPT# value in table DEPT (to reflect the fact that every employee must be assigned to an existing department).

Columns DEPT# in table DEPT and EMP# in table EMP are the *primary keys* for their respective tables. Column DEPT# in table EMP is a *foreign key,* referencing the primary key of table DEPT.

2.1.1 The SQL Language

One of the most important ways to manipulate data in a relational database is through the declarative query language *SQL* (Standard Query Language).
Most current relational products support some dialect of SQL. SQL was originally developed by IBM Research in the early 1970s, it was first implemented on a large scale

in the IBM relational prototype System R and subsequently reimplemented in numerous commercial products from both IBM and other vendors. Dialects of SQL has since become an American (ANSI) national standard, an international (ISO) standard, a UNIX (X/Open) standard, and an IBM standard.

SQL is used to formulate relational operations (i.e., operations that define and manipulate data in relational form). Let’s begin with defining the departments-and-employees database:

```
CREATE table DEPT(
   DEPT# char(2),
   DNAME char(20),
   BUDGET decimal(7),
   PRIMARY KEY (DEPT#));

CREATE table EMP(
   EMP# char(2),
   ENAME char(20),
   SALARY decimal(5),
   DEPT# char(2),
   PRIMARY KEY (EMP#),
   FOREIGN KEY (DEPT#) REFERENCES DEPT);
```

Having created the tables, we can start operating on them by means of the SQL data manipulation operations SELECT, INSERT, UPDATE and DELETE. In particular, we can perform relational SELECT, PROJECT and JOIN operations on the data, in each case using the SQL data manipulation statement SELECT. See the examples below.

Example 1: SELECT (RESTRICT)

```
SELECT DEPT#, DNAME, BUDGET
FROM DEPT WHERE BUDGET > 8M;
```

DEPT#	DNAME	BUDGET
D1	Marketing	10M
D2	Development	12M

Table 2-3. Result of example 1.

Example 2: PROJECT

```
SELECT DEPT#, BUDGET FROM DEPT;
```

DEPT#	BUDGET
D1	10M
D2	12M
D3	5M

Table 2-4. Result of example 2.

Example 3: JOIN

```
SELECT DEPT.DEPT#, DNAME, BUDGET, EMP#, ENAME, SALARY
FROM DEPT, EMP
WHERE DEPT.DEPT# = EMP.DEPT#;
```

DEPT#	DNAME	BUDGET	EMP#	ENAME	SALARY
D1	Marketing	10M	E1	Jones	40K
D1	Marketing	10M	E2	Ewing	30K
D2	Development	12M	E3	McCoy	45K
D3	Research	5M	E4	Chekov	25K

Table 2-5. Result of example 3.

Note that the join example above (Example 3) illustrates the point that qualified names (e.g. DEPT.DEPT#, EMP.EMP#) are sometimes necessary in SQL to “disambiguate” column references. If unqualified names were used, that is, if the WHERE clause were of the form “WHERE DEPT# = DEPT#, then the two “DEPT#” references would be ambiguous (it would not be clear in either case whether the reference stood for DEPT.DEPT# or EMP.DEPT#).

A query language is not called relationally complete unless it provides at least the three basic operations selection, projection and join. Selection produces a subset of the rows of a table, the preceding queries do this by specifying constraints on the tables 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 records in two different tables that have equal or related values in the specified attributes.

2.2 Object-oriented databases

Today’s relational products are inadequate in a number of ways, and maybe the relational model is inadequate too. Some of the features that seem to be needed in DBMSs have existed for many years in object-oriented programming languages. Thus, it is only natural to investigate the idea of incorporating those features into database systems, and hence to consider the possibility of object-oriented database systems.
The basic idea of OO database systems is similar to that of OO programming languages: Users should not have to wrestle with computer-oriented constructs such as bits and bytes (or even records and fields), but rather should be able to deal with objects and operations on those objects, that more closely resemble their counterparts in the real world. For example, instead of having to think in terms of a “DEPT tuple” plus a collection of corresponding “EMP tuples” that include “foreign key values” that “reference” the “primary key value” in that “DEPT tuple”, the user should be able to think directly of a *department object* that actually contains a corresponding set of *employee objects*. And instead of having to “INSERT” a “tuple” into the “EMP relation” with an appropriate “foreign key value” of some “tuple” in the “DEPT relation”, the user should be able to *hire* an *employee* object directly into the relevant *department* object. In other words, the fundamental idea is to **raise the level of abstraction**.
Naturally, raising the level of abstraction is a desirable goal, and the OO paradigm has been very successful in meeting that goal in the programming languages arena. Therefore, it is natural to ask whether the same paradigm can be applied in the database arena also. The idea of dealing with a database that is made up of *encapsulated objects* (e.g. objects that “know what it means” to hire an employee or change their manager or cut their budget), instead of having to understand relations, tuple updates, foreign keys, etc., is naturally much more attractive from the user’s point of view.

Object-oriented DBMSs are expected to meet the requirements of new application domains, such as:

- computer-aided design and manufacturing (CAD/CAM)
- computer-integrated manufacturing (CIM)
- computer-aided software engineering (CASE)
- geographic information systems (GIS)
- science and medicine
- document storage and retrieval

All of the above represent areas in which today’s relational products tend to run into trouble.

Thus, the fundamental concept and modelling construct in an OO DBMS is the concept of *object*. Objects are used to model physical or abstract entities in the domain of interest. Every object has a *type* (the OO term is *class*). Individual objects are sometimes referred to as object instances specifically, in order to distinguish them clearly from the corresponding object type or class.

All objects are *encapsulated*. This means that the representation (i.e. the internal structure) of a given object is not visible to users of that object, instead users know only that the object is capable of performing certain functions (methods). For example, the methods that apply to DEPT objects might be HIRE_EMP, FIRE_EMP, CUT_BUDGET, etc. The advantage of encapsulation is that it allows the internal representation of objects to be changed without requiring any of the applications that use those objects to be rewritten. In other words, encapsulation implies *data independence*.

Every object has a unique identity called its “object ID” or *OID*. Primitive objects like the integer 5 are self-identifying, i.e., they are their own OIDs; other objects have (conceptual) addresses as their OIDs, and these addresses can be used elsewhere in the database as pointers to refer to the objects in question. One implication of this is that objects does not necessarily have to have any user-defined candidate keys as in a relational system.

OO concepts typically involve some generalization mechanism for types, providing capabilities to structure types into hierarchies.
First, object class Y is said to be a *subclass* of object class X, equivalently, object class X is said to be a *superclass* of object class Y, if and only if every object of class Y is necessarily an object of class X (“Y **ISA** X”). Objects of class Y then *inherit* the instance variables and methods that apply to class X. As a consequence, the user can always use a Y object wherever a X object is permitted and thereby take advantage of *code reusability*. The ability to apply different methods with the same name to different classes is referred to as *polymorphism*. Some systems also support the notion of *multiple inheritance*, in which a given class can be a subclass of several classes simultaneously.

3 The AMOS DBMS

3.1 AMOS II

AMOS (Active Mediators Object System) is a research DBMS prototype which conforms to systems classified as object-relational DBMSs. AMOS is a main-memory DBMS and is therefore very fast compared to disk-based DBMSs. AMOS has been developed built on substantial developments of the WS-Iris main-memory object-oriented DBMS engine [9]. WS-Iris was developed at Hewlett-Packard Laboratories and is a derivative of Iris [10].
The AMOS architecture uses the mediator approach [11] that introduces an intermediate level of software between databases and their use in applications. Each AMOS server has DBMS facilities, such as a local database, a data dictionary, a query language, transaction processing and remote access to databases. The query language, AMOSQL, is a derivative of OSQL [12]. AMOSQL extends OSQL with active rules, a richer type system and multidatabase functionality [8]. AMOS II is the latest generation of the AMOS system, running on the Windows NT platform.

3.2 The mediator approach

The mediator approach [11] introduces an intermediate level of software between databases and their use in applications and by users. The purpose of a mediator is to query, monitor, transform, combine and locate desired information between a set of applications and data sources. An external data source can be a conventional DBMS, data files with specific exchange file formats or other mediators (In this case other AMOS II servers). Image 3-1 shows an example of an AMOS II mediator system in which some applications access data sources through a mediator system. The mediator presents high-level abstractions (views) of combinations of these data sources. Notice that the mediator can access another AMOS II server as a data source.

Translators implements the mapping between local schemas (in the data sources) and the corresponding component schemas (in the common data model). Thus, there is one translator for every kind of data source. A query sent to a translator is transformed into calls to the underlying data source. The results of these calls are then processed to form an answer to the initial query.

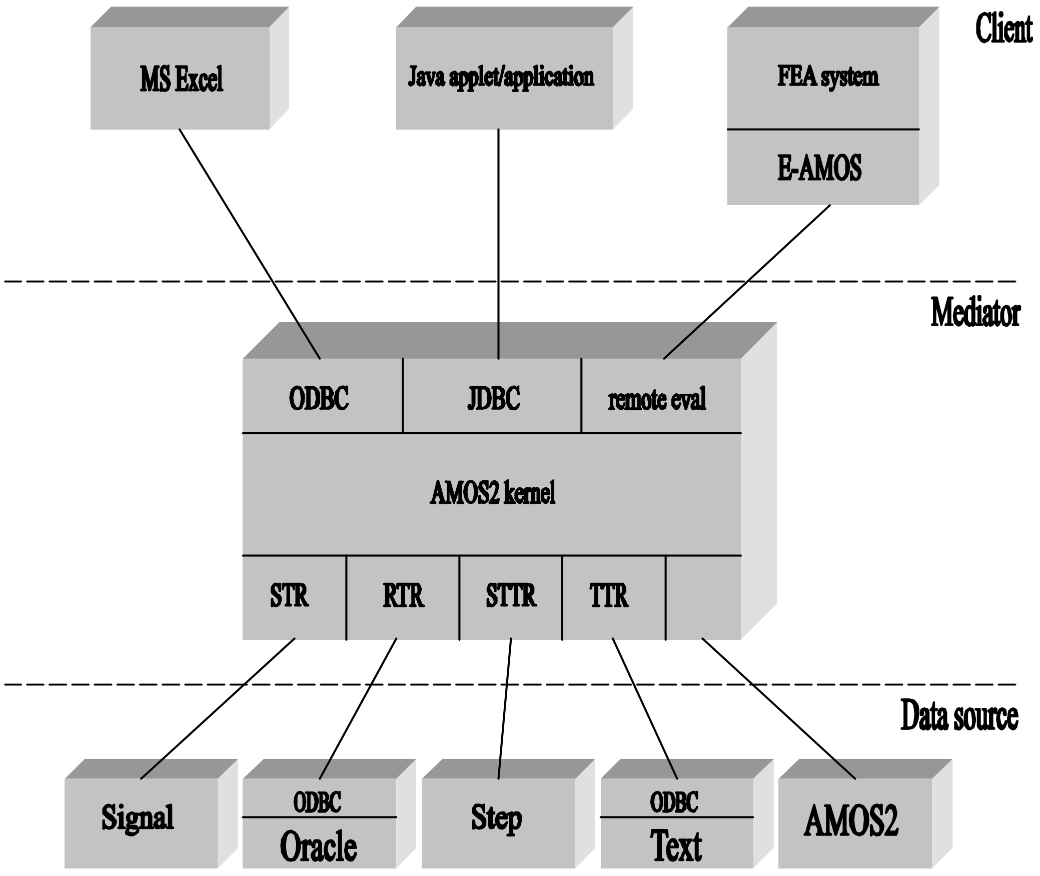


Image 3-1. 3-level multidatabase architecture.

3.3 The AMOSQL language

AMOSQL is a functional language with object-oriented extensions. The language is more than relationally complete. Its basic capabilities include constructs for database schema definition and evolution, population and updates, and database queries in terms of the basic data model that includes objects, types and functions. Furthermore, it supports logical operators, arithmetic operators, active rules, multidatabase queries, disjunctive queries, quantification, nested subqueries, transitive closures and so on. AMOSQL provides a declarative query language interface to the database. This nature of the language requires that optimization of queries is performed before execution can take place.

3.3.1 The AMOS data model

The data model consists of the basic constructs *objects*, *types* and *functions*. There are two types of objects in AMOS. *Literal objects*, such as *character string*, *integer*, *real*, *boolean*, etc. are self defining. The other type is called a *surrogate object*, these objects have unique object identifiers (*OIDs*). Surrogate objects represent physical or abstract external or internal concepts, e.g. persons, houses, doors or whatever might be in the database. System-specific objects such as types and functions are also treated as surrogate objects.
Types are used to structure objects according to their functional characteristics, in other words, objects can be structured into types. Types are themselves related in a type hierarchy of subtypes and supertypes. That is, subtypes inherit functions from their supertypes, and they can even have multiple supertypes. In addition, functions can be overloaded on different subtypes (having different implementation for different types).
Functions are defined on types, and are used to represent attributes of, relationships among and operations on objects. Functions can be defined as stored, derived, procedure or foreign. A stored function has its extension explicitly stored in the database, while a derived, procedure or foreign function has its extension defined in an AMOSQL query, an AMOSQL procedure or a function in an external language such as Lisp or C.
Furthermore, functions can be overloaded, i.e. functions defined for different combinations of arguments can have the same name (*polymorphism*). The selection of the correct function implementation of an overloaded function is made at function invocation based on the actual argument types.

3.3.2 AMOSQL examples

AMOSQL provides statement constructs for typical database tasks, such as data definition, population, updates, querying and transaction control. Data schemas can be defined, modified and deleted by using AMOSQL statements. The definition of types, functions and objects is performed through the `create` statement. For example, types may be defined by a `create type` statements as:

```
create type named_object;


create function name(named_object) -> charstring as stored;


create type person subtype of named_object;
```

where two types, named_object and person are defined. A new type becomes an immediate subtype of all supertypes provided in the `subtype` clause, or if no supertypes are specified, it becomes a subtype of the system type `UserTypeObject`. In the example above, since person is a subtype of `named_object`, it inherits the property function `name` (defined on `named_object`). The same thing can also be accomplished by:

```
create type named_object properties (name charstring);


create type person subtype of named_object;
```

A database is populated with objects with a `create` statement, with or without initializations of functions. For example like this:
First, let’s add a function to the type `person`:

```
create function age(person) -> integer as stored;
```

Then, populate the database with:[1]

```
create person(name, age) instances
:p1(‘Sara’, 26),
:p2(‘Marcus’, 27);
```

Derived functions are defined in a similar way as stored functions. A single AMOSQL query is in the function body. For example like this:

```
create function person_older_than(integer a) -> charstring
as select name(p) from person p where age(p) > a;
```

Let’s add one more function to our example database:

```
create function friends(person) -> bag of person;
```

The type `bag` holds the result of queries as sets of objects with duplicates retained.
In addition to population by object creation and attribute assignment, it is possible to use the function update statements set, add and remove. Example:

```
set friends(:p1) = :p2;
```

Deletion of types, functions and objects can be made through the `delete` statement as:

1. Variables preceded by a colon, such as :p1, are global variables used by AMOSQL to hold results temporarily during a session.

```
delete :p1;

delete function name;

delete type person;
```

3.4 AMOS II External Interfaces

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 functions through the *callout* interface. Currently there are interfaces between AMOS II and the languages C and Lisp, while other interfaces are being developed (Java) [2].

3.4.1 The callout interface

In the callout interface, the AMOS II kernel calls external functions written in Java, C or Lisp. Essentially, foreign AMOSQL functions are implemented by a number of external functions which can be defined through a special mechanism called *the multi-directional foreign function* interface [2]. Foreign functions in AMOSQL must be side-effect free since the query optimizer may rearrange their calling sequence.
The callout interface has similarities with *data blades* or *data cartridges* in Object-Relational databases. The system also allows the callin interface to be used by the callout interface, which gives great flexibility. Of course, the callout interface always runs in the same address-space as AMOS II.

3.4.2 The callin interface

In the callin interface a program, written in C, calls AMOS II. This interface is similar to the call level interfaces for relational database systems, such as Oracle or Sybase.
The two basic alternatives for connecting applications to AMOS II are either through a *tight* or a *loose* connection. In the tight connection, or *embedded AMOS II connection,* AMOS II is directly linked together with the C-based application. This means that the application and AMOS II runs in the same address-space and therefore this provides the fastest connection possible. By using a driver program in C that initializes AMOS II and catches AMOS II errors, the DBMS can be linked to the application as a C-library. An obvious disadvantage with this tight connection is that execution errors in the application may cause AMOS II to crash. Another disadvantage is that only a single application can be linked to AMOS II, which means that AMOS II becomes a single application system.

For the loose connection, or *client-server connection*, the application can work as a client to AMOS II. In this case several applications can access the AMOS II server concurrently. In this situation, the application and the AMOS II server are executing in different processes, possibly on different machines. This approach makes the AMOS II server more resistant to execution errors in the application. If a run-time error occurs, it will not affect the AMOS II server. The main disadvantage of this approach is the overhead of inter-process communication. In comparison to the tight connection, the access time can be several orders of magnitude higher in this loose connection.
For both types of connections there are two possible ways to communicate with AMOS II from the host language of the application, either through the *embedded query* interface or through the *fast-path* interface.

- In the embedded query interface strings containing AMOSQL statements are passed to AMOS II for evaluation. Primitives for accessing the result of the AMOSQL statements from C are provided. The embedded query interface is relatively slow since the AMOSQL statements have to be parsed and compiled before execution.

- In the fast-path interface predefined AMOS II functions are called from the C program, without the overhead of parsing and compiling. Therefore, the fast-path interface is significantly faster than the embedded query interface. Of course, this assumes that the AMOSQL function is already defined in the database. If it is not predefined, it has to be defined and then called. In that case it is actually slower than the embedded query interface, since the query has to be parsed, compiled and then run. However, if the same function will be called again, perhaps with different arguments, the fast-path interface is much faster since the query only has to be parsed and compiled once.

Thus, the conclusion is:
If the statement should only be executed once and is not already defined in the database, use the embedded interface. Otherwise, always use the fast-path interface.

4 ODBC

This chapter describes the background and architecture of ODBC.

4.1 Introduction

While many standards have been proposed to address the needs of multi-DBMS (Database Management System) access, ODBC has emerged as the de facto standard for the MS Windows platform and is a component of Microsoft's Windows Open Services Architecture (WOSA). Essentially all modern RDBMSs (Relational Database Management System) products support this standard, either as their sole interface to the outside world or in addition to their own proprietary interface.

4.1.1 The evolution towards ODBC

In the early days of data processing technology, there was no such thing as a database management system. Each vendor's system came with a proprietary file access system that was unique to the machine's hardware and operating system. The file access systems were implemented primarily in software. All were record oriented – read and write operations were performed on a single record at a time. Code developed for one environment could not be migrated to another without major revisions to the source code, the file access routines, and the operating system interface components of the application. Therefore applications were developed exclusively for each hardware platform and seamless portability and interoperability were only a dream.

Into this world of incompatible file systems came the database management system. In file access systems, each file was treated as a separate, stand-alone entity, with no inherent relationships to any other file. There was no central place where information about that file was stored. It was up to the programmers to tell the file access system what a record in each file looked like by providing a record definition section in their programs.

Gradually DBMSs evolved to address both of these two major flaws of file system implementations. System files or tables were maintained by the DBMS that described all data elements (tables, indexes, columns, etc.) and the relationships among them. This information about the data is referred to as metadata, and it paved the way for the implementation of some of the basic data integrity constraints that we take for granted today. Referential integrity constraints, for example, were not available in early file systems, as each file was a separate unit unconnected with the outside world. The metadata maintained by a DBMS treats many tables as interrelated units, with their relationships identifiable through primary and foreign key linkages.

Of course, early DBMSs were not relational. They still required programmers to be skilled in the navigation schemes used by the DBMS engine. It was not sufficient to specify what information you wished, you also had to specify the access method that the DBMS should use to retrieve that information. This required highly skilled programmers for each DBMS interface and end user access to information was impossible without intervention of the programming staff.

The rules of relational theory by Dr. E.F. Codd in the late 1970s laid the groundwork for the evolution of the relational databases we have today. The mathematical foundation of relational algebra and calculus provided the framework upon which a structured query language (SQL) could be developed. SQL could ease the burden of figuring out how to retrieve the data, freeing programmers to worry only about what information they needed, not how to get it. The programmers were given a single common DBMS interface when SQL was accepted by all RDBMS vendors.

But compatibility and proprietary interface problems continued to plague the industry, mostly because of the advent of client/server computing and the necessity to interface the RDBMS to the application via a proprietary network operating system (e.g. Novell). In the RDBMS world of the late eighties and early nineties, each application continued to be developed to perform a specific task and to work with a single database engine.

Additionally, SQL was not complete enough that the RDBMS vendors could use it without their own extensions. In the rush to distinguish its products by the addition of advanced features, each vendor expanded the basic SQL with its own proprietary enhancements to support the capabilities offered by its products.

By this time the SQL Call Level Interface (*SQL/CLI*) emerged as an attempt to address the new interface component introduced by the network operating system and to resolve incompatibilities between the SQL dialects offered by competing vendors. The SQL/CLI is designed to support database access from shrink-wrapped applications and was originally created by a subcommittee of the SQL Access Group (*SAG*[1]). The SAG/CLI[2]

1. SAG, SQL Access Group. A group of database vendors creating a standard for remote database access.
2. Actually, the new term for the SAG/CLI is X/Open CLI (based on SQL89).

specification was published as the Microsoft Open Database Connectivity specification in 1992.

4.2 What is ODBC?

Microsoft developed the ODBC interface as a means of providing applications with a single application programming interface (API) through which to access data stored in a wide variety of DBMSs. ODBC is designed to give applications the ability to access different database management systems with the same source code. The data source is not necessarily a DBMS, an application can even access text-files or Excel documents using ODBC. Today ODBC is a very widespread API with hundreds of ODBC-enabled applications.

4.2.1 ODBC architecture

The question then, is how does ODBC standardize database access? There are two architectural requirements:

- Applications must be able to access multiple DBMSs using the same source code without recompiling or relinking.

- Applications must be able to access multiple DBMSs simultaneously.

Then there is one more question, due to marketplace reality:

- Which DBMS features should ODBC expose? Only features that are common to all DBMSs or any features that is available in any DBMS?

The problem is solved in the following way:

- *ODBC is an application programming interface.* To solve the problem of how applications access multiple DBMSs using the same source code, ODBC defines a standard API. This contains all of the functions in the CLI specifications from X/Open[1] and ISO/IEC[2] and provides additional functions commonly required by applications.

  A different library, or *driver*, is required for each DBMS that supports ODBC. The

1. The Open Group, a group “committed to lower the barriers of integrating new technology across the enterprise.” Sponsored by Compaq, Fujitsu, HP, Hitachi, IBM, NCR, Siemens, Sun and many more.
2. The International Organization for Standardization and the International Electrotechnical Commission.

driver implements the functions in the ODBC API. To use a different driver, the application simply loads the new driver and calls functions in it. To access multiple DBMSs simultaneously, the application loads multiple drivers.

- *ODBC defines a standard SQL grammar.* In addition to a standard API, ODBC defines a standard SQL grammar. This grammar is based on the X/Open SQL CAE specification.

  Applications can submit statements using ODBC- or DBMS-specific grammar. If a statement uses ODBC grammar that is different from DBMS-specific grammar, the driver must convert it before sending it to the data source. However, such conversions are rare since most DBMSs already use standard SQL grammar.

- *ODBC provides a Driver Manager to manage simultaneous access to multiple DBMSs.* Although the use of drivers solves the problem of accessing multiple DBMSs simultaneously, the code to do this may be complex. Applications that are designed to work with all drivers can not be statically linked to any drivers, instead they must load and unload drivers dynamically at runtime and call functions in them through a table of function pointers. Naturally, this situation becomes more complex if the application uses multiple drivers simultaneously. The Driver Manager helps the application to do all this.

- *ODBC exposes a significant number of DBMS features but does not require drivers to support all of them.* If ODBC exposed only features that are common to all DBMSs, it would be of little use. After all, the main reason so many different DBMSs exist today is that they have different features. ODBC only requires that drivers implement a subset of all those features.

The ODBC API architecture varies according to the operating system. On the Windows platform, ODBC uses a dynamic-link library (DLL) architecture with loadable database drivers and a Driver Manager. The Windows implementation of ODBC is quite similar to the Windows print model, where the application developer writes to a generic printer interface and a loadable driver maps that logic to hardware-specific commands.

The ODBC architecture has four components (see Image 4-1):

- **Application:** Performs processing and calls ODBC functions to submit SQL statements and retrieve results.

- **Driver Manager:** Loads and unloads drivers on behalf of the application. Processes ODBC 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 the syntax supported by the associated DBMS.

- **Data Source:** Consists of the data the user wants to access and it's associated DBMS.

Multiple drivers allow the application to simultaneously access multiple data sources. The ODBC API is used in two places: between the application and the driver manager and between the driver manager and the driver.

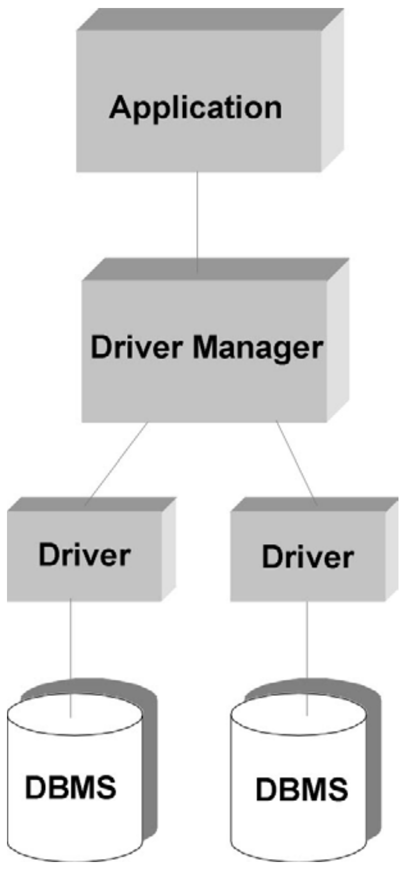


Image 4-1. Components of the ODBC architecture.

4.2.2 The Driver Manager

The Driver Manager is a library that administers communications between applications and drivers. On the Windows platform, the Driver Manager is a DLL written by Microsoft. The Driver Manager takes care of common problems, such as determining which driver to load, loading and unloading drivers and calling functions in drivers. The Driver Manager is either statically linked to the application, or loaded by the application at runtime. The application calls ODBC functions in the Driver Manager, not the ODBC driver. Therefore, the Driver Manager must implement all ODBC functions. Mostly, this is done as a simple pass-through call to the function in the correct driver, but the Driver Manager also implements a few functions, such as functions for getting information about installed drivers, which ODBC-functions a driver supports, etc. It also performs some basic error checking and maps deprecated functions to guarantee backward compatibility of ODBC 3.0 drivers that are used with ODBC 2.x applications. If the application calls an ODBC 2.x function that is not implemented in the ODBC 3.0 driver, the Driver Manager calls the corresponding ODBC 3.0 function instead. Therefore, it is important that if the driver uses ODBC 3.0, then the Driver Manager must be of at least version 3.0. (It is always possible to use older applications and drivers with a newer Driver Manager.)

4.2.3 Drivers

The driver is a library that implements the functions in the ODBC API. Each driver is specific to a particular DBMS. The driver exposes the capabilities in the corresponding DBMS. If, for example, the DBMS does not support outer joins, then neither should the driver.

Some tasks performed by the driver include:

- Connecting and disconnecting from data sources.
- Checking for function errors not checked by the Driver Manager.
- Initiating transactions.
- Submitting SQL statements to the data source for execution. The driver must of course modify ODBC SQL to DBMS-specific SQL (In this case AMOSQL).
- Sending and receiving data from the data source, including converting data types.
- Mapping DBMS-specific errors to ODBC errors (ODBC SQLSTATEs).

There are two kinds of drivers, file-based drivers and DBMS-based drivers. File-based drivers access the database file directly (for example a driver for a simple text-file). In this case the driver acts as both driver and data source, it processes ODBC calls and SQL statements. This kind of driver has to implement its own database engine. DBMS-based drivers, on the other hand, access the physical data through a separate database engine.

The driver only processes ODBC calls, SQL statements are passed to the database engine for processing, that is, the driver acts as the client in a client/server configuration where the DBMS acts as the server.

4.2.4 Conformance Levels

Naturally, there is no agreement on what is the proper set of functionality of a DBMS. Even databases that follow the relational model, uses SQL as its query language and run on client-server architecture have no consensus on functionality. Different DBMSs naturally have different functionality and users purchase the product partially due to the extended functionality that the product offers.
To handle different database functionality, ODBC provides a minimum level of functionality that is expected to be supported by all drivers, while still utilizing as many features of a DBMS as possible. Application developers have to decide whether to use the minimum level of functionality or to test for extended functionality.

Many DBMSs have sets of functionality in common, and therefore ODBC defines different conformance levels, both for the API and for SQL statements. Each driver lets the application determine at runtime what ODBC capabilities and what SQL grammar the driver and each data source supports.

In ODBC 3.0, drivers are classified based on what features they possess. Three levels are defined (Core, Level 1 and Level 2), and to meet a particular conformance level a driver must satisfy all of the requirements of that level. However, conformance levels do not divide neatly into support for a specific list of ODBC functions. To support a feature, the driver must support some or all forms of calls to certain ODBC functions, setting certain attributes and certain descriptor fields. Drivers are free to implement features beyond the level to which they claim conformance. Applications can discover any such additional capabilities by calling **SQLGetFunctions** (to determine which ODBC functions are available) and **SQLGetInfo** (to query various other ODBC capabilities).

4.3 Basic flow of an ODBC application

This part describes how an ODBC-based application connects to a data source, submits queries and fetches results.

First of all, the application (or user) selects which data source to connect to. Then the driver manager loads the correct driver for that data source. The application starts by asking the driver to allocate memory for an environment (which is driver-specific) and return a environment handle to the application. This handle is used in subsequent calls to the driver. Next, a connection to the data source itself has to be established. For this, a connection handle is needed. When a connection has been established, a statement handle for sending queries and retrieving results is needed. All three types of handles are driver specific, the application only uses a handle (an address) in subsequent calls to the data source.

Queries to the data source can be submitted in two different ways, either direct execution or prepared execution. Direct execution is the simplest option. Just send a query to the data source for execution, for example a simple select statement. No parameters are allowed, that is, all necessary data has to be sent at once. This type of execution might be appropriate for simple queries which are expected to be executed only once. Since the data source has to compile the query before execution, this method is rather slow, at least if the same query is asked many times. In this case, prepared execution is significantly faster.

For prepared queries, the query is sent to the data source for preparation (compilation). The query can be executed at a later time. It is also possible to send parameters to the data source before execution. An example: The select statement

```
SELECT age FROM person WHERE name = ?
```

where ‘?’ is a parameter marker might be sent to the data source. Before execution, the value of the parameter is sent. After retrieving the result of the query new parameters can be sent for the next execution of the query. The query never has to be compiled again, at least not until its associated statement handle has been freed.

Retrieving of results is done in a loop (fetch the next tuple if there is one) and at the same time type conversion is made if necessary.

When the application wishes to disconnect from the data source all statement handles are freed (a connection can have many concurrent statements), the driver disconnects from the data source, and the connection and environment handles are freed. At this point the application is completely disconnected from the data source.

4.4 The future of ODBC

ODBC has been a great success, at least on the Windows platform. There are hundreds of ODBC-enabled applications, for example: MS Excel, Word, Access, Internet Information Server and Filemaker Pro. The future may be a bit more uncertain. It seems as if Microsoft is phasing out ODBC. Apparently, Microsoft wants us to use OLE DB instead. The latest version of ODBC (3.5) includes support for OLE DB. ODBC 3.5 adds unicode support and includes the OLE DB Provider for ODBC Drivers as a part of the Driver

Manager. This means that all OLE DB applications can use existing and future ODBC drivers.

4.4.1 OLE DB

Cite from Microsoft’s OLE DB web page*: “OLE DB is Microsoft's strategic low-level interface to data across the organization. OLE DB is an open specification designed to build on the success of ODBC by providing an open standard for accessing all kinds of data.”*

OLE DB is a set of interfaces for data access that provides universal data integration regardless of the data type. The ODBC data access interface will continue to provide a unified way to access relational data as part of the OLE DB specification. However, in the future Microsoft expects OLE DB to lead new database products that are assembled from best-in-class components rather than from the monolithic products available today. OLE DB is supposed to provide an efficient and flexible database architecture that offers applications, compilers and other database components efficient access to Microsoft and third-party data stores.

OLE DB is the fundamental *Component Object Model* (COM) building block for storing and retrieving records from databases. It will be used throughout Microsoft’s future line of applications.

An OLE DB data provider exposes data from an underlying data source. For example, the OLE DB Provider for ODBC exposes ODBC data sources, and the OLE DB Provider for Jet exposes Microsoft Jet (the underlying DBMS used by Access). An OLE DB data consumer is something that consumes the data exposed by a provider.

5 Driver design

The ODBC standard specifies what an ODBC driver should do and what functions it must export. However, it does not specify *how* this should be done. Therefore different driver architectures were proposed and rejected. This chapter shows different possible architectures of the AMOS II ODBC-driver and finally the chosen architecture.

5.1 Possible driver architectures

The driver is a DLL which is being loaded into the client application's addresspace. It should export all necessary ODBC functions for the conformance level it claims conformance to. The SQL statements being transmitted from the application must be converted to the data source specific query language, in this case AMOSQL.
The usual architecture of a DBMS-based driver is for the driver to be completely standalone from the DBMS (See image 5-1). The driver sends SQL-statements to the DBMS using the network and receives data in the same way. Usually the application/driver runs on a different machine than the DBMS, or at least in another process.

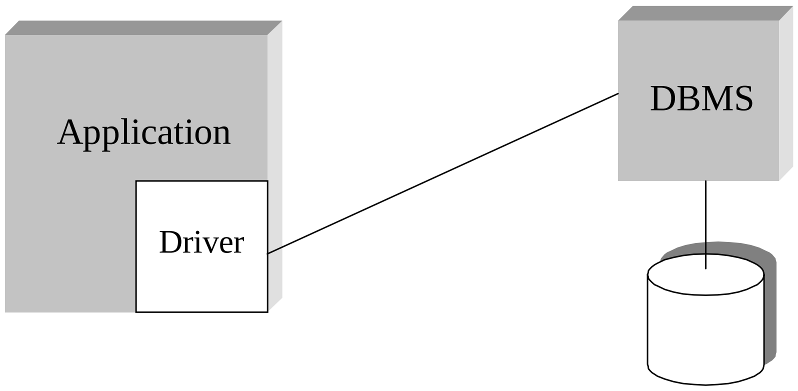


Image 5-1. The usual architecture of a DBMS-based ODBC-driver.

This architecture was of course the first considered. Since AMOS II currently lacks an easy-to-use client-server interface, it would be necessary to implement such an interface. The AMOS II server would in this case run in nameserver mode and the driver would connect to the port that server listens to. The driver would send AMOSQL queries to the AMOS II server and receive data which must be parsed. Thus, a parser must be developed for converting SQL-statements to AMOSQL-statements and another parser would translate the data received from the AMOS II server to a form suitable for the driver.

The next architecture considered was a small variation of the first. Suppose that we split the driver in two. The parsing of SQL to AMOSQL could be done on the server side. This could be implemented as an external function to AMOS II using the *callout* interface. By using this architecture, AMOS II would suddenly have a SQL interface aswell as an AMOSQL interface (See image 5-2). The SQL interface could of course be used by other applications than the driver. The driver would still have to parse the data received from AMOS II, unless of course this too would be done on the server side. Still this would require a lot of work. A variation on the same approach could be to do all the work on the server side, using the *callout* interface (See image 5-3). In this case the implementation of the ODBC functions in the driver would mostly be pass-through calls to the server. This could really be viewed as if the driver was completely on the server-side since all work would be done there. This might seem as a strange approach to the problem, but in this way the driver would have complete access to AMOS II.

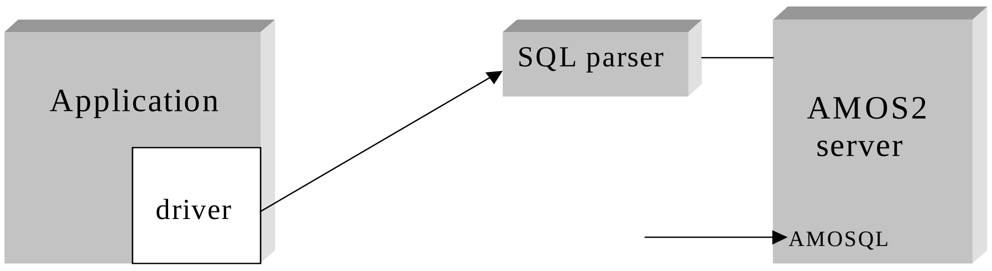


Image 5-2. The AMOS II server could listen for connections on two different ports.

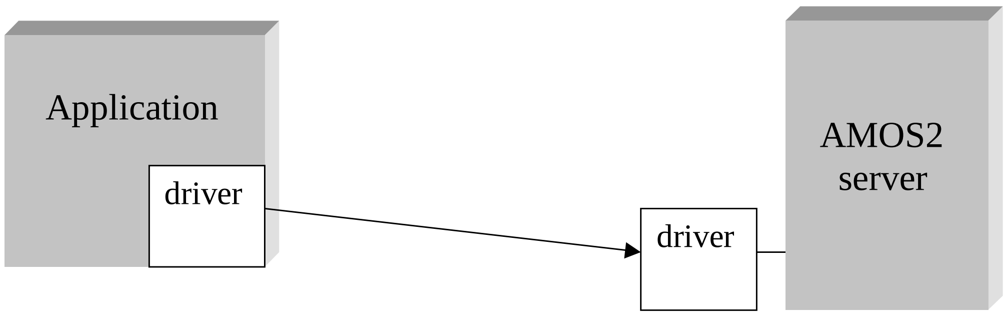


Image 5-3. Driver on the server side. The client side driver passes through ODBC calls to the server.

5.2 The chosen architecture

The last proposal was to use the *callin* interface to embed the AMOS II system into the driver. Since AMOS II would be statically linked to the driver and the application dynamically loads the driver, the entire AMOS II system would be embedded into the application e.g. Excel! (See image 5-4). Of course, we were really excited about this possibility. The ODBC connection to AMOS II would be ultra-fast, since the system would run in the same address space as the application and the fact that AMOS II is a main memory system. The downside would be the fact that embedded AMOS II is a single user system. The application would have it’s own local, private database. As will be shown, there is a way around this problem (The embedded AMOS II system can connect to other AMOS II servers, running on other machines.) Another disadvantage is the fact that the driver would be a lot larger than an ordinary ODBC driver. AMOS II uses about 3 MB of memory, but with a large database it will be a lot more.

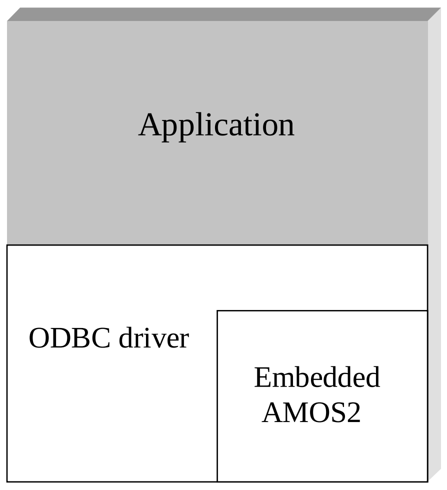


Image 5-4. AMOS II embedded in the ODBC driver.

6 Implementation

Implementing an ODBC-driver consists mostly of writing C-code, which is not very interesting to read about. Instead, this chapter covers parts of the implementation process which perhaps differs somewhat from the development of an “ordinary” relational database driver. In our case, the relational model of ODBC has to be converted to the object model of AMOS. Moreover, the SQL queries from ODBC must be translated into AMOSQL queries.

6.1 Data model mapping

At a first glance, AMOS II and ODBC may seem mismatched. The data models appear to be completely different with a relational model on the ODBC side and an object model on the AMOS II side. The ODBC driver has to solve this problem. The object model from the AMOS II database must be mapped into a transient model for ODBC. As an example, consider that a three dimensional object (like the Earth) can be represented in two dimensions on paper (a map). In essence, the projection is accomplished by “flattening” the object model into tables that have relationships through foreign keys.

- **Object Identifier.**
  The object identifier (OID) of each object becomes the primary key for each object. This key can be used to build relationships with other tables.

- **Inheritance**
  Since inheritance has no counterpart in the relational world, the AMOS II driver simply maps derived types into separate tables. For example, a type *employee* that is derived from the type *person* is represented as an *employee* table containing all data from the *person* part plus all data from the *employee* part. Of course, this creates some redundancy in the mapping since the *person* part of the *employee* object is also contained in the *person* table.

6.1.1 The POET database system

POET Software is a german software company which developed the POET Object Server Suite, an object database for Windows NT.
Quote:
“POET Software is the leading provider of high performance, scalable object databases for Windows NT and Java, providing unique value in embedded applications that require small footprint and zero management overhead.”

Since there is an ODBC-driver for POET, let’s take a look at POETS’s way of mapping the object model into a relational model.

First, consider a very simple example where there is only one persistent class defined in the database. This class has three data members which hold, respectively, the name, age and weight of a child. Each object in the database corresponds to one child and holds three basic items of data.

```
persistent class Child
{
   public:
      char name[30];
      short age;
      float weight;
};
```

The class Child is mapped as a table CHILD, which has four columns:

name	**age**	**weight**	**child_oid**

Table 6-1. The CHILD table.

The last column corresponds to the object identification in the POET database. Each row has a unique value for column CHILD_OID, this value serves as the primary key for the table, which is sometimes required for updating. The value cannot be updated; every newly added row gets a unique value provided by the POET database, which never repeats that from a previously deleted row. A data type is associated with each column. The ODBC driver must allocate an ODBC data type to every column of every table.

Column	ODBC data type
name	SQL_CHAR
age	SQL_INTEGER
weight	SQL_FLOAT
child_oid	SQL_INTEGER

Table 6-2. Child data types.

The application program must then map each of these SQL data types to a type appropriate for its own needs. This trivial example illustrates the basis that every persistent class is mapped as a basetable, which is then visible through an ODBC application such as Microsoft Access. Often, however, the one-to-one mapping just described is not adequate, additional tables must be generated. Three commonly used features of the POET paradigm require a more complex approach:

- a data member is a data aggregate rather than primitive.
- a data member is a set (collection of objects).
- a data member is a reference to another object.

Aggregates are expanded

POET treats the members of an aggregate as though they were members of the object containing it. As an example consider the class Address defined as an aggregate of street, city, zip, state:

```
persistent class Address
{
   public:
      char street[20], city[20], zip[6], state[3];
};
```

Objects of class Person contains the address as a data member:

```
persistent class Person
{
   public:
      char name[30];
      Address address;
};
```

The table PERSON would then be structured as:

name	address_street	address_city	address_zip	address_state	person_oid

Table 6-3. The PERSON table.

If the embedded object contains further embedded objects, these will also be expanded, using the aggregates membername as above. This way of expanding aggregates may seem smart for simple examples as the one above. However, consider a slightly more advanced type:

```
persistent class Person
{
   public:
      char name[20];
      int age;
      Person friend;
};
```

In this example the type Person contains a data member (*friend*) which also is of type *friend*. Since aggregates are expanded, the member *friend* would be expanded to *friend_name*, *friend_age* and so forth. But this *friend* will also be expanded! This way, the member type *friend* might be infinitely expanded (at least until there is a person in the database without a friend).

6.1.2 Mapping AMOS II databases

For simple types, like the CHILD type above, the AMOS II ODBC-driver performs exactly the same mapping as the POET ODBC-driver.

```
create type CHILD
   properties (name charstring, age integer, weight real);
```

The CHILD type is mapped as:

OID	name	age	weight

Table 6-4. The CHILD table.

Unlike the POET ODBC-driver, the AMOS II ODBC-driver does not expand aggregates. Instead an aggregate of the kind discussed above becomes a foreign key to another table (actually the OID in another table).

```
create type ADDRESS
   properties (street charstring,
   city charstring, zip charstring, state charstring);

create type PERSON
   properties (name charstring, address ADDRESS);
```

In this case the PERSON table is mapped as:

OID	name	ADDRESS_OID

Table 6-5. The PERSON table.

Types containing sets of elements (bag of) is mapped as two different tables, one containing the object and one containing the OID of the object and a column containing all values in the bag. See the following example:

```
create type car_manufacturer
   properties (name charstring, models bag of charstring);
```

The type car_manufacturer is mapped as two separate tables with OID as primary/foreign keys. See table 6-6 and 6-7.

OID	name

OID	models

Table 6-6 and 6-7. The type car_manufacturer mapped as two separate tables.

Types containing a function returning more than one value would be possible to expand within the same table as the rest of the type, but since this would violate the relational model, it will be mapped to a separate table containing the OID of the object and a column for every result of the function. See the example below where the function *data* returns two values:

```
create type person
  properties (name charstring);

create function data(person) -> <integer, real>;
```

OID	name

OID	data_1	data_2

Tables 6-8 and 6-9. The type person mapped as two separate tables.

6.2 Parsing SQL statements

In programs with structured input, such as an ODBC-driver, two tasks that appear over and over are dividing the input into meaningful units, and then discovering the relationship among the units. For a SQL-statement, the units are variable names, strings, operators, punctuation, and so forth. This division into units (which are usually called *tokens*) is known as *lexical analysis*.
As the input is divided into tokens, the relationship among the tokens has to be established. This task is known as *parsing* and the list of rules that define the relationships that the program understands is a *grammar*.

Of course, the SQL statements generated by an ODBC-enabled application has to be converted into it's AMOSQL counterpart. For example[1] the SQL query (generated by Microsoft Query):

```
SELECT TEACHER.NAME, DEPT.NAME
FROM DEPT DEPT, TEACHER TEACHER
WHERE TEACHER.WORKS_AT = DEPT.OID
```

Must be translated into:

```
select name(teacher), name(dept)
from dept dept, teacher teacher
where works_at(teacher) = dept;
```

As can be seen in this case, there is not a very big difference between the SQL- and AMOSQL-syntax (at least not for SELECT statements).
Some queries may use variables, which are sent at a later time. Naturally, these queries must also be translated correctly. Example:

```
SELECT STUDENT.NAME, STUDENT.EMAIL, STUDENT.MAJOR
FROM STUDENT STUDENT
WHERE (STUDENT.MAJOR<>?)
```

Where '?' is a parameter marker.
Is translated to:

```
select name(student), email(student), major(student)
from student student where (major(student)!=?1);
```

In the AMOS II ODBC-driver, parameters are numbered as ?1, ?2, and so on.

1. See appendix E for a definition of the classes used in these examples.

The code for translating SQL queries into AMOSQL queries was generated using the GNU[1] tools *Flex* and *Bison*. Flex takes a set of descriptions of possible tokens and produces a C-routine, a *lexical analyser*, that can identify those tokens. (Flex is the GNU version of the standard UNIX tool Lex). Bison is a general-purpose parser generator that converts a grammar description for a *LALR* (*L*ook*A*head *L*eft *R*ecursive)[7] context free grammar into a C program to parse that grammar. Bison is upward compatible with *Yacc* (yet another compiler compiler).

The Flex specification used to generate the lexical analyser and the Bison grammar description can be found in Appendix A and B, respectively.

Since many ODBC-applications allows the user to write their own SQL-statements, this opens up a possibility to bypass the translation of SQL to AMOSQL. By prefixing the statement to be executed with *amosql:* the translation is skipped. Thus, it is possible to write your own queries using AMOSQL instead of SQL. This is actually even used internally in the driver when calling stored functions (see the next chapter).

6.3 Using stored procedures

Some ODBC functions are best implemented as stored AMOS procedures[2]. Functions specifically suited for this are those for getting metadata such as all available tables (SQLTables), columns (SQLColumns), data types (SQLGetTypeInfo) and so on. For example, see the function for getting information about supported data types (SQLGetTypeInfo) below. The stored procedures *ODBCGetTypeInfoSpecificType* and *ODBCGetTypeInfoAllTypes* are called for getting information on a specific data type and all supported data types respectively. The procedures are actually executed using another ODBC-function, SQLExecDirect. Later, SQLFetch is called to fetch the results of the executed query.

1. GNU’s Not Unix! See www.gnu.org
2. See appendix C for all stored procedures and stored data for ODBC support.

```
EXPORT SQLRETURN SQL_API SQLGetTypeInfo(
  SQLHSTMT StatementHandle,
  SQLSMALLINT Datatype)
{
  STMT* stmt=(STMT*)StatementHandle;
  char errstr[MAX_STRING_LENGTH];

  delete_error_records(&stmt->diagnostics);
  change_header_record(&stmt->diagnostics, 0, "", 0, 0);

  /* This query is a TYPE_INFO query. Needed by SQLFetch. */
  stmt->query_type=TYPE_INFO;

  switch (DataType)
  {
    case SQL_INTEGER:
    case SQL_DOUBLE:
    case SQL_VARCHAR:
    {
      char buffer[50];
      sprintf(buffer,"%sODBCGetTypeInfoSpecificType(%i);",
              AMOSQL_TAG,DataType);
      /* Call stored function */
      return SQLExecDirect(stmt, buffer, strlen(buffer));
    }
    case SQL_ALL_TYPES:
    {
      char buffer[50];
      sprintf(buffer, "%sODBCGetTypeInfoAllTypes();", AMOSQL_TAG);
      /* Call stored function */
      return SQLExecDirect(stmt, buffer, strlen(buffer));
    }
    default:
    {
      char* state="HY004";  /* Invalid SQL data type */
      sprintf(errstr, "%sInvalid SQL data type", DRIVER_ERROR_STR);
      EnterCriticalSection(&stmt->lock);
     post_error_record(&stmt->diagnostics,XOPEN,SQL_NO_COLUMN_NUMBER,
       SERVER_NAME,errstr,0,SQL_NO_ROW_NUMBER,DBMS_NAME,state,XOPEN);
      LeaveCriticalSection(&stmt->lock);
      DEBUG("SQLGetTypeInfo returned SQL_ERROR");
      return SQL_ERROR;
    }
  }
}
```

A few help functions also had to be implemented, for example one for getting all attributes for a specific type. This translates to all columns of a table in our relational model. This is one of the two functions actually written in Lisp:

```
(create-function user_attributes ((type t)) ((function f))
   as (f)
   foreach ((type tp))
   where (and (= tp (allsupertypes t))
                  (!= tp  (typenamed “object”))
                  (= f (attributes tp))))
```

That is, get all attributes of the type not inherited from the type *object*.

6.4 Executing SQL statements

This chapter describes how the driver implements query execution and retrieving of results from executed statements.

6.4.1 Preparing queries for execution

Prepared queries are the fastest way to execute the same query many times, maybe with different variables for every execution. The query is prepared (compiled) in the data source and executed at a later time. The query remains active until the corresponding statement environment is freed, or in this case until disconnection from AMOS.

The function for preparing queries, *SQLPrepare*, takes a statement handle and a statement text as input. The statement text has to be converted to AMOSQL, this is of course accomplished by a call to the bison-generated parsing procedure. The parser returns the translated statement and the number of arguments to that statement.

After parsing, a *transient* function has to be generated. A transient function is an AMOS function which remains accessible until the database is shut down.
The AMOSQL query preparation (PREPARE-QUERY QUERY ARITY) generates a transient function. The actual code for generating a function for later use looks like this: (All called functions are from the AMOS callin interface.)

```
a_setf(fun, a_getfunction(env->connection,
       "charstring.integer.prepare_query->function", FALSE));
a_setarity(argl, 2); □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
a_setstringelem(argl, 0, Text, FALSE);   □□□□□□□□□□□□□□□□□□□□□□□□□□□
a_setintelem(argl, 1, num_params, FALSE);
a_callfunction(env->connection, s, fun, argl, TRUE);

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
a_getrow(s, row, FALSE);
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
a_setf(stmt->fun, a_getelem(row, 0, FALSE));
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
a_setarity(stmt->argl, num_params);
free_tuple(argl);
free_tuple(row);
free_oid(fun);
free_scan(s);
```

where the variable *fun* temporarily stores a reference to the generated function. The function reference is then stored in the currently active statement's function variable (*stmt->fun*) for later use. The variable *Text* holds the statement text, for example *select name(p) from person p where age(p) > ?1*. The variable *num_params* holds the number of arguments in the statement text.

Some applications (Microsoft Office) seems to be unable to remember what statements it just prepared, because it calls SQLPrepare before every execution of the same statement. Therefore, the just prepared query (actually the SQL statement text) is cached. Before parsing the text the current statement text is compared to the cached text. If the texts match, parsing and query preparation can be avoided.

6.4.2 Binding parameters

Before executing a prepared query argument values, if any, must be supplied. For every bound column an *Application Parameter Descriptor* (APD) and an *Implementation Parameter Descriptor* (IPD) is created, if they were not created previously. The descriptors contain information about the bound columns, such as column number and data type. If the data types in the IPD and the APD differ (for the same column) that means that the application wants the driver to convert the supplied parameter value from the type supplied to the type at the data source. All descriptors are stored in a simple linked list in the current statement environment.

6.4.3 Executing prepared statements

Before executing the prepared statement the parameters (see above) are converted if necessary and then the previously stored function is executed. See the code for *double* type parameters below:

```
case SQL_DOUBLE:
{
  double data;
  if(!C_to_SQL(APD_record->type,APD_record->data_ptr,SQL_DOUBLE,
     &data))
  {
    char* state="07006";
    sprintf(errstr,"%sRestricted data type attribute violation",
            DRIVER_ERROR_STR);
   post_error_record(&stmt->diagnostics, XOPEN, SQL_NO_COLUMN_NUMBER,
                      SERVER_NAME, errstr, 0, SQL_NO_ROW_NUMBER,
                      DBMS_NAME, state, XOPEN);
    goto error;
  }
  a_setdoubleelem(stmt->argl, (current->id)-1, data, FALSE);
  break;
}
```

As can be seen from the code above, the parameter is converted to the correct data type (the call to C_to_SQL) and then added to the argument list. The call to the AMOS callin interface function *a_setdoubleelem* adds the argument value (data) to the argument list (stmt->argl), where the argument number is given by the expression *(current->id)-1* (-1 because AMOS numbers parameters from 0 and ODBC from 1.)

When all parameters has been added to the argument list the function is executed:

```
a_callfunction(env->connection,stmt->scan,stmt->fun,stmt->argl,TRUE)
```

After execution, the number of columns in the result is checked and for each column a new *Implementation Row Descriptor* (IRD) is created. Just as for APD’s and APD’s, the IRD contains information about data types and names of the columns in the result set.

6.4.4 Direct execution

For direct execution, the SQL statement is parsed, just as for prepared execution, but then it is executed directly using this function call:

```
a_execute(env->connection, stmt->scan, Text, TRUE)
```

where the variable *Text* contains the AMOSQL statement.

After execution, the number of columns in the result set is investigated and new IRDs are created.

```
a_openscan(env->connection, stmt->secondary_scan, FALSE)
a_getrow(stmt->secondary_scan, stmt->tuple, TRUE)
num_cols=(SQLSMALLINT)a_getarity(stmt->tuple, FALSE);
```

As can be seen from the code above, a secondary scan is opened only for checking the number of columns in the result set. The first row in the set is retrieved but not used. After getting a row from the result set, the row pointer is advanced one step. Later, when retrieving data from the result, a new scan is opened, pointing to the first row.

6.4.5 Binding result columns

Before actually retrieving the results of an executed query, the application has to bind all columns it is interested in getting results from. Not all columns are necessarily bound. Upon execution, the driver creates IRDs (one IRD for every column in the result) with column information. For every column the application is interested in, an *Application Row Descriptor* (ARD) is created. An ARD contain information on what data type the application expects the driver to convert the result to and an address to where the driver should store the converted data. For string types, the size of the allocated space is also stored in the ARD.

6.4.6 Retrieving results

When retrieving data from the result set, the application calls SQLFetch for every row to get data from. The driver examines all bound columns (all ARDs) and retrieves data for those columns, converting the data if necessary.

7 Problems

This chapter describes some of the problems encountered during the work.

7.1 Getting information on how to implement a driver

The main problem was definitely finding information on how to implement a driver. All books I could find on the subject mainly described how to develop an ODBC-based application. Even Microsoft’s book on the subject, *Microsoft ODBC 3.0 Programmer’s Reference and SDK Guide*, had little information on driver responsibilities. My solution to the problem was to “reverse engineer” the information found in the literature and to trace ODBC function calls from applications such as Microsoft Query and Access. Some hints was also found when studying the source code from an existing ODBC driver for the MySQL DBMS published by T.c.X Datakonsult AB. Unfortunately, their driver was an ODBC 2.5 driver, while the AMOS driver I was developing was version 3.0.
Because of these rather fundamental problems, the development process took far more time than expected.

7.2 Testing the driver

During development of the driver, I had to find out a way to test the driver. Since ODBC applications can only use completely implemented drivers (others will simply be rejected) I had to develop my own ODBC application capable of testing my incomplete driver. At first I did not compile the driver as a DLL, instead I developed an application calling my functions directly. Few errors were discovered using this technique since both the exported ODBC functions and the application were implemented using my own view of how an ODBC application/driver should work. Other (real) applications might expect different things from the driver.

Later in the development process, I began testing the driver using Microsoft’s test tool *ODBC Test*. This application makes it possible to test individual calls to an unfinished

driver. This was an excellent test tool, but there were still problems. As soon as the function SQLFetch was called, the application crashed. This meant that I could test most of the features of the driver except the most important one. I spent several weeks searching for the problem without success.
After implementing a few more ODBC functions it could be used by a real ODBC application, Microsoft Query. To my surprise SQLFetch worked without any problems.

7.3 Debugging a DLL

When testing the driver during longer sessions it would sometimes crash, some errors seemed regular and some not. To find these bugs, a nice tool for debugging the driver was needed. Unfortunately, he tool used for developing, *Borland C++ Builder*, was unable to debug DLLs. The excellent (but somewhat buggy) debugger in *Microsoft Visual Studio* on the other hand is very capable of debugging DLLs. Unfortunately the problem was that nobody could figure out what calling convention[1] to use when compiling with Visual Studio. When compiling with C++ Builder the convention specified in ODBC 3.0 was used (__*stdcall*). However, this did not work when compiling with Visual Studio. Perhaps Microsoft changed the meaning of *stdcall* when developing Visual Studio?

Description of __*stdcall* according to Microsoft:

Called function pops its own arguments from the stack.
An underscore (_) is prefixed to the name.
The name is followed by the at sign(@) followed by the number of bytes (in decimal) in the argument list.
Therefore, a function declared as
```
int foo(int a, double b)
```

is decorated as follows:

```
_foo@12
```

Description of __*stdcall* according to Borland:

Called function pops the stack.
Case is preserved.
Does *not* generate underscores.

1. Calling convention options tell the compiler which calling sequences to generate for function calls. The different calling conventions differ in the way they handle stack cleanup, order of parameters, case and prefix of global identifiers.

Unfortunately, this meant that it was impossible for us to use the debugger in Visual Studio.

Most of the “regular” bugs have been corrected, but there still are some problems. A reason for these problems might be the multi-threaded properties of the application. The driver should be fairly thread-safe, but as even the standard C-library may have some flaws in respect to this, I can’t be really sure.

7.4 Stack overflow

On some occasions the application would crash with a stack overflow when executing queries. The size of the C-stack is fixed and decided by the application, the user has no way of setting the stack size. Many functions in the AMOS system is recursive (Lisp) and thus needs a lot of stack space. The problem was fixed by “warming up” (running a few AMOS functions from the command-line) the database image before use with an ODBC application.

8 Using the driver - a demonstration

This chapter contains an example when using the driver with an ODBC application, in this case Microsoft Query.

After connecting to the data source, the application has retrieved all tables and lets the user choose what tables and columns the query should consist of. (Image 8-1)

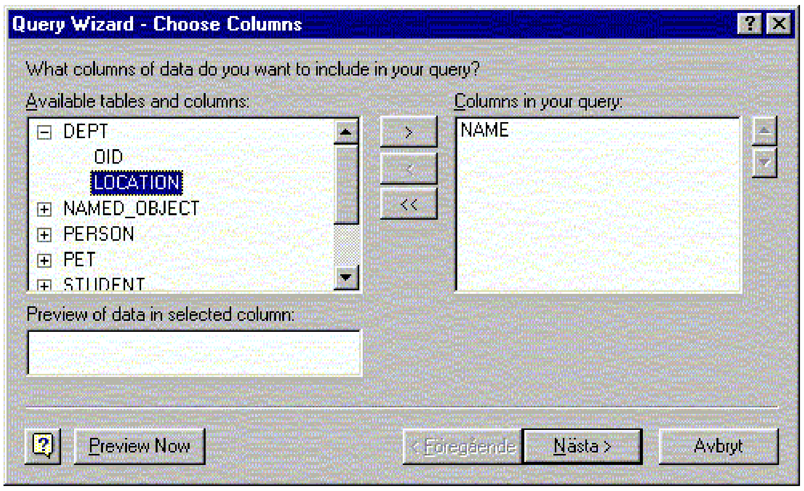
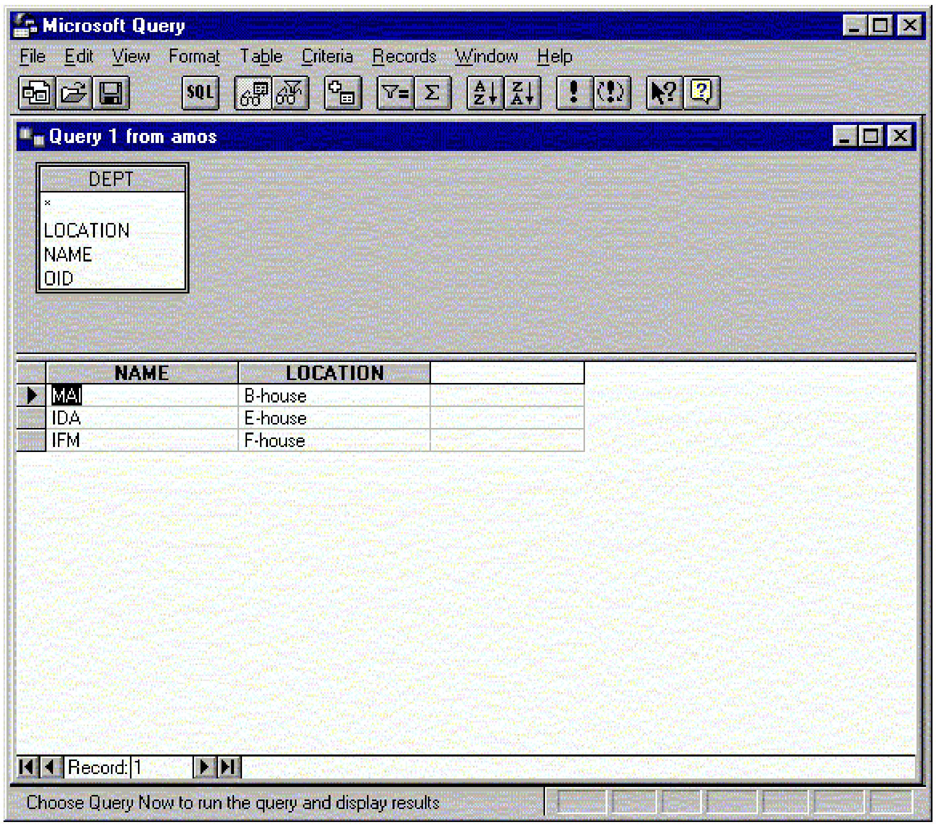

Query Wizard - Choose Columns
What columns of data do you want to include in your query?
Available tables and columns:
DEPT
OID
LOCATION
NAMED_OBJECT
PERSON
PET
STUDENT
Columns in your query:
NAME
Preview of data in selected column:
Preview Now
< Föregående
Nästa >
Avbryt

Image 8-1. Adding tables and columns to the query.

When the query has been designed it is executed. In this case we wanted all department names and their locations.


Microsoft Query
File Edit View Format Table Criteria Records Window Help
SQL
Query 1 from amos
DEPT
*
LOCATION
NAME
OID

NAME	LOCATION
MAI	B-house
IDA	E-house
IFM	F-house

Record: 1
Choose Query Now to run the query and display results

Image 8-2. The result of the executed query.

Since this query wasn't very interesting, let's add the *teacher* and the *subject* table to it:

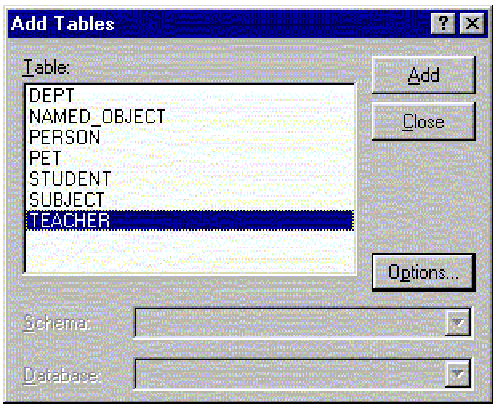
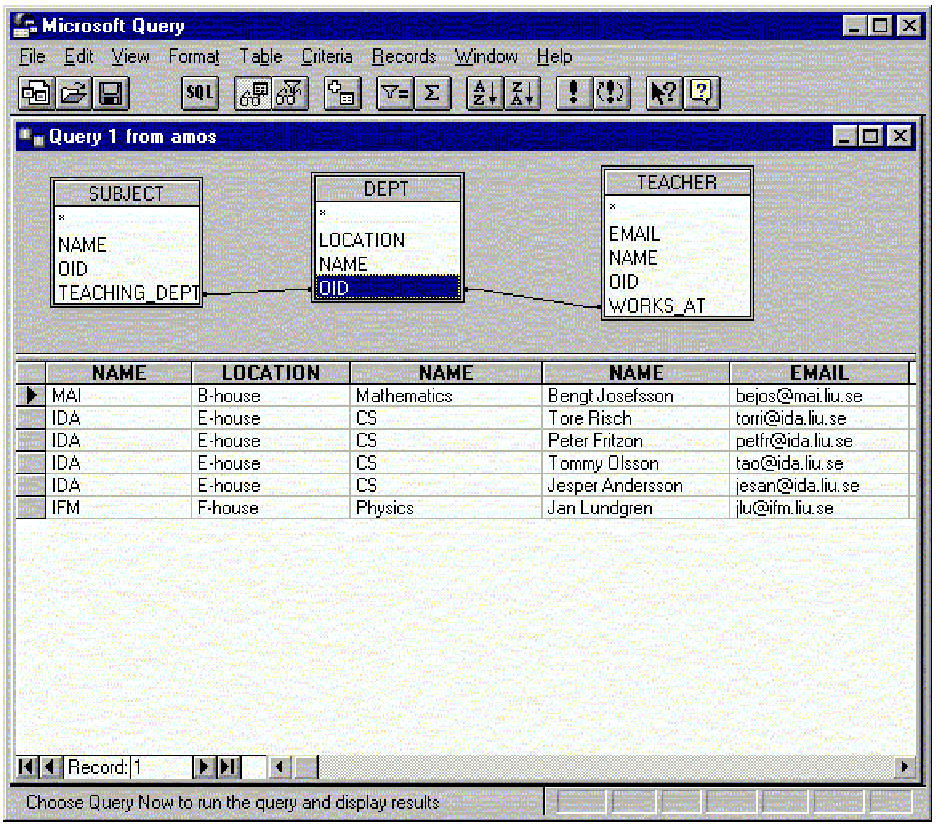


Image 8-3. Adding more tables to the query.

Then we join the three tables together as can be seen in the image below:

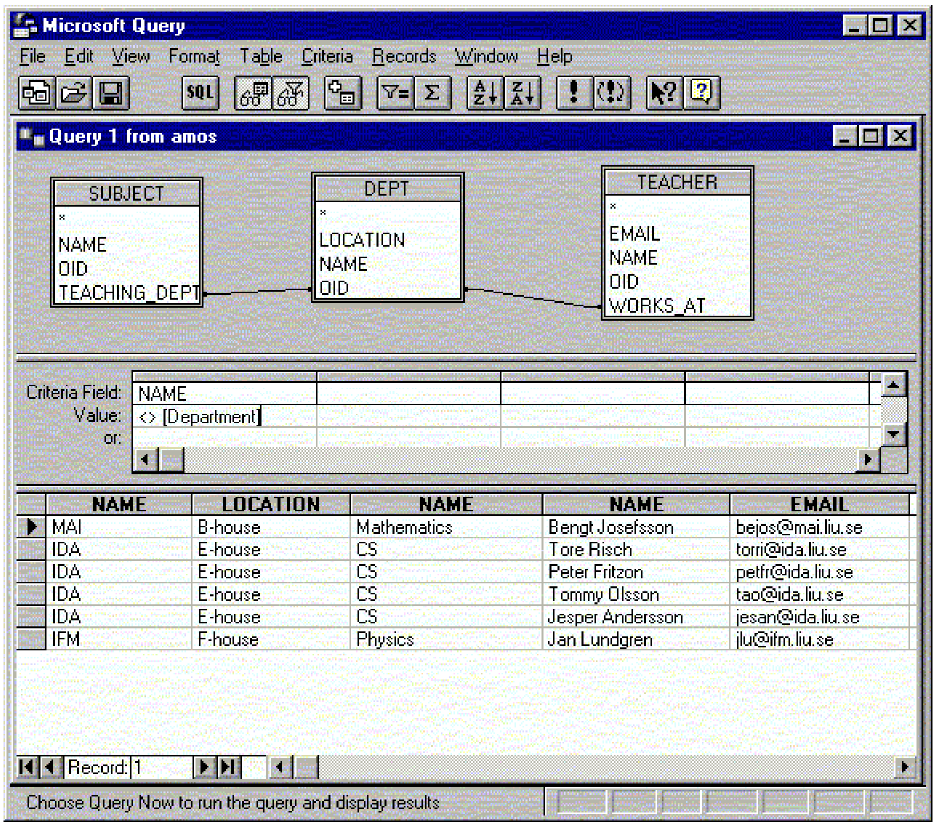


NAME	LOCATION	NAME	NAME	EMAIL
MAI	B-house	Mathematics	Bengt Josefsson	bejos@mai.liu.se
IDA	E-house	CS	Tore Risch	torri@ida.liu.se
IDA	E-house	CS	Peter Fritzon	petfr@ida.liu.se
IDA	E-house	CS	Tommy Olsson	tao@ida.liu.se
IDA	E-house	CS	Jesper Andersson	jesan@ida.liu.se
IFM	F-house	Physics	Jan Lundgren	jlu@ifm.liu.se

Choose Query Now to run the query and display results

Image 8-4. Result of the query after joining on department.

Now, assume we do not want the teachers at a specific department. We add a criteria to the query. This will generate a query with a parameter (*dept.name*).

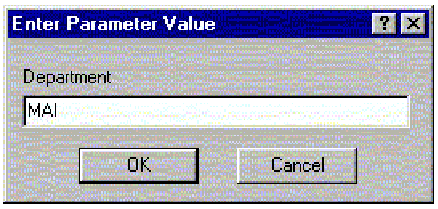


Criteria Field:	NAME
Value:	<> [Department]
or:	

NAME	LOCATION	NAME	NAME	EMAIL
MAI	B-house	Mathematics	Bengt Josefsson	bejos@mai.liu.se
IDA	E-house	CS	Tore Risch	torri@ida.liu.se
IDA	E-house	CS	Peter Fritzon	petfr@ida.liu.se
IDA	E-house	CS	Tommy Olsson	tao@ida.liu.se
IDA	E-house	CS	Jesper Andersson	jesan@ida.liu.se
IFM	F-house	Physics	Jan Lundgren	jlu@ifm.liu.se

Image 8-5. Adding a parameter to the query.

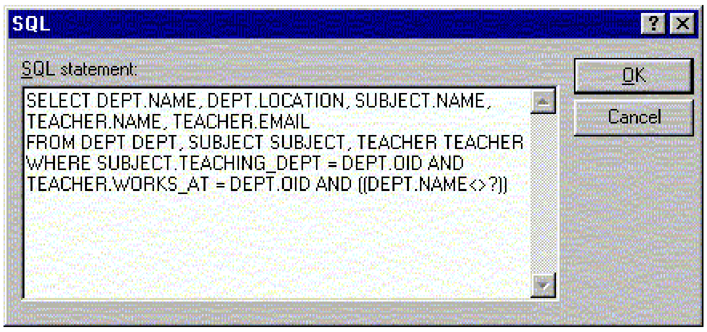
On execution, we have to enter the value of all parameters. In this case we are not interested in teachers working at MAI.



Department: MAI

Image 8-6. Entering the value of the parameter before execution.

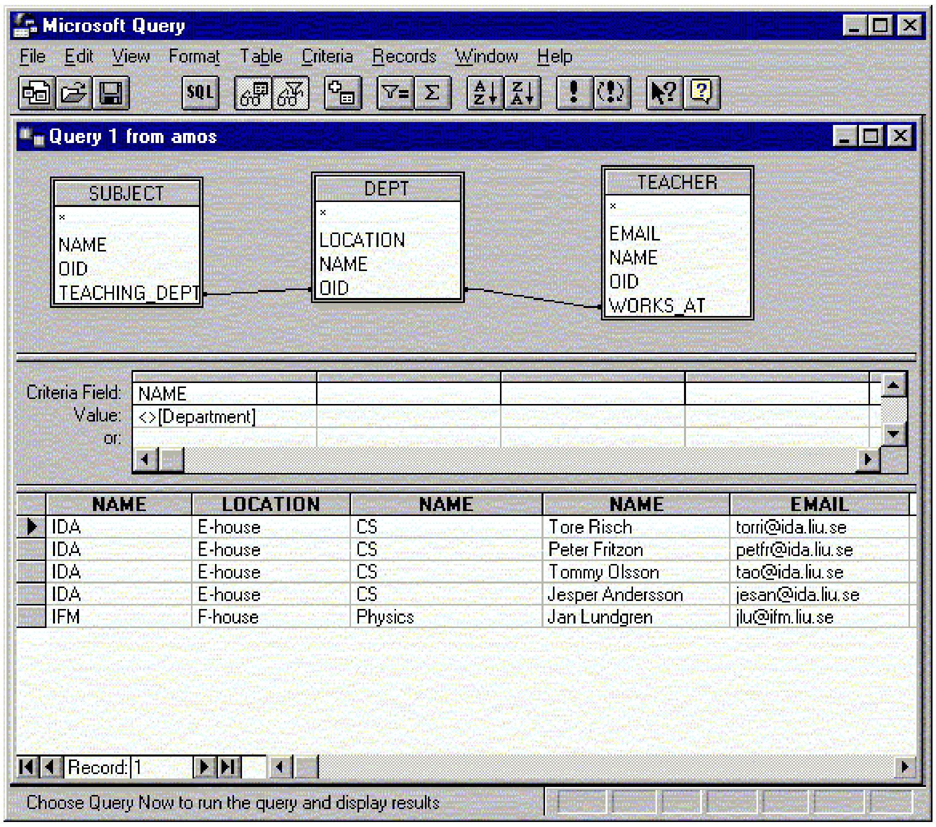
The generated SQL statement looks like this:



```
SELECT DEPT.NAME, DEPT.LOCATION, SUBJECT.NAME,
TEACHER.NAME, TEACHER.EMAIL
FROM DEPT DEPT, SUBJECT SUBJECT, TEACHER TEACHER
WHERE SUBJECT.TEACHING_DEPT = DEPT.OID AND
TEACHER.WORKS_AT = DEPT.OID AND ((DEPT.NAME<>?))
```

Image 8-7. Generated SQL query using an argument. This will be parsed into AMOSQL by the driver.

And the result is of course the same as before, but without MAI:

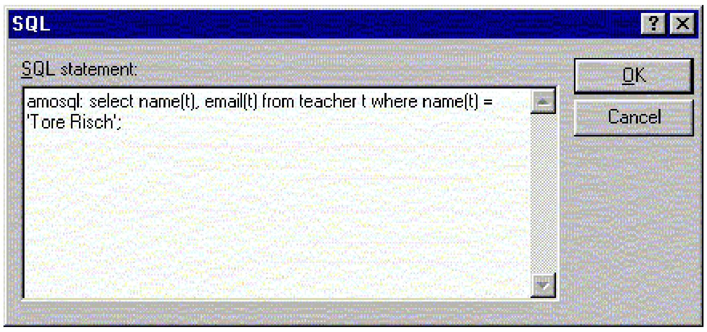


Criteria Field:	NAME
Value:	<>[Department]
or:	

NAME	LOCATION	NAME	NAME	EMAIL
IDA	E-house	CS	Tore Risch	torri@ida.liu.se
IDA	E-house	CS	Peter Fritzon	petfr@ida.liu.se
IDA	E-house	CS	Tommy Olsson	tao@ida.liu.se
IDA	E-house	CS	Jesper Andersson	jesan@ida.liu.se
IFM	F-house	Physics	Jan Lundgren	jlu@ifm.liu.se

Image 8-8. Result of the generated query.

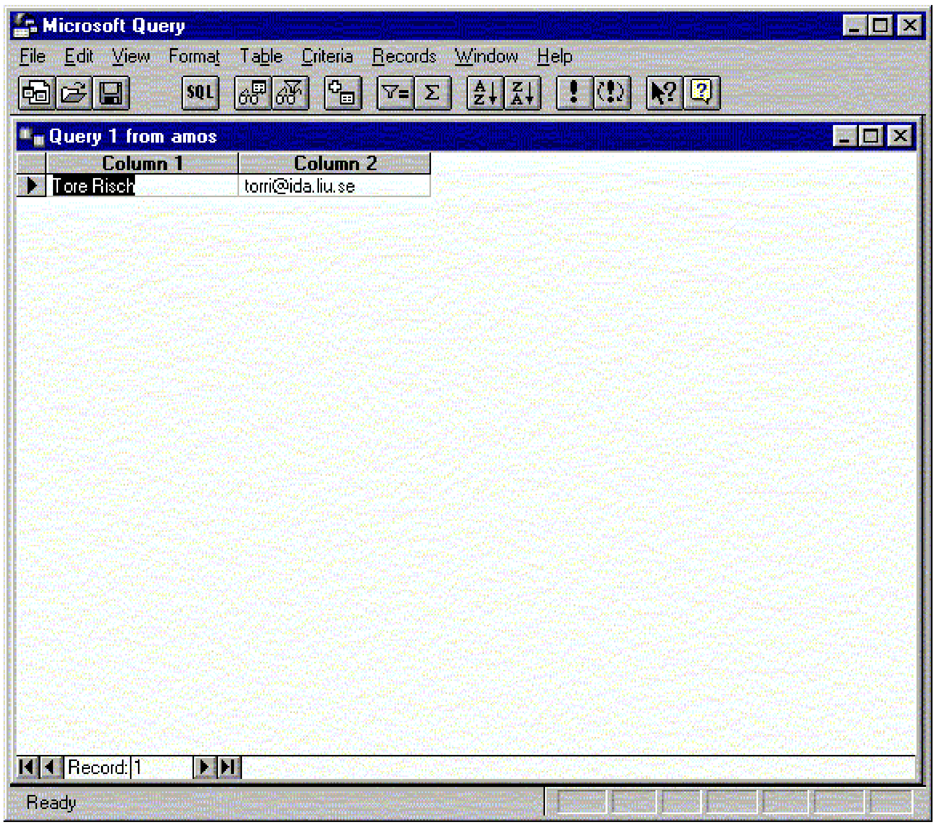
Since we can edit our own queries (for users familiar with SQL), we can write our own queries using AMOSQL. Just prefix the statement with *amosql:*.



```
amosql: select name(t), email(t) from teacher t where name(t) =
'Tore Risch';
```

Image 8-9. Entering an AMOSQL query prefixing with *amosql:*.

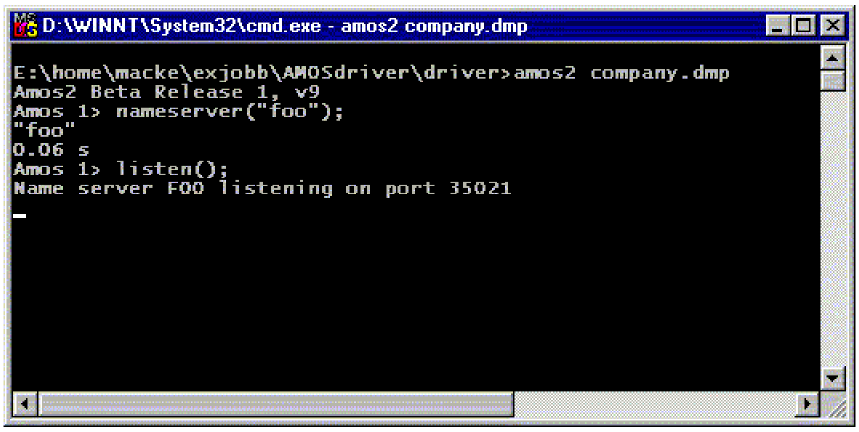
As can be seen in the result below, the driver does not use correct column names. For SQL statements generated the usual way in MS Query, the application does not use the column names from the driver. Instead it is using the column names from the generated SQL statement. Naturally, this does not work when the user writes his own AMOSQL statements. This is one of the things missing in the driver. (See the next chapter on future work.)



Column 1	Column 2
Tore Risch	torri@ida.liu.se

Image 8-10. Result of the executed AMOSQL query.

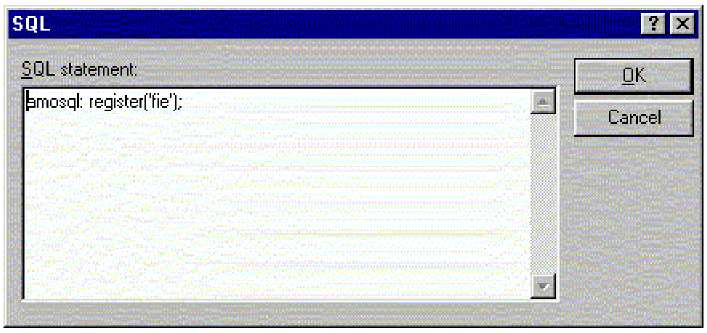
To demonstrate that the driver really is very different from ordinary ODBC drivers, we start a new example using AMOSQL queries. First, start an AMOS server on some machine:



```
E:\home\macke\exjobb\AMOSdriver\driver>amos2 company.dmp
Amos2 Beta Release 1, v9
Amos 1> nameserver("foo");
"foo"
0.06 s
Amos 1> listen();
Name server FOO listening on port 35021
```

Image 8-11. Starting an AMOS server using the *company* database.

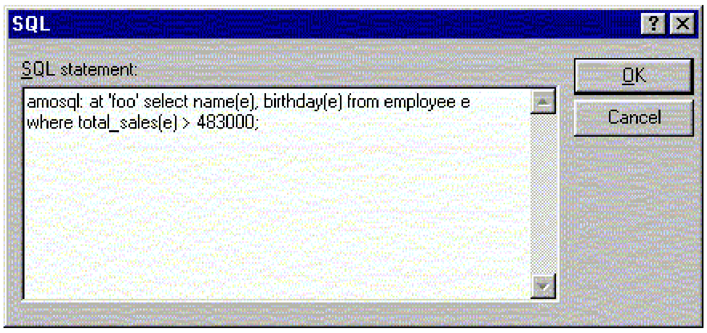
Now an AMOS server named “foo” is waiting for connections. Now, connect the ODBC application to this server:



```
amosql: register('fie');
```

Image 8-12. Connecting the ODBC application to the AMOS server.

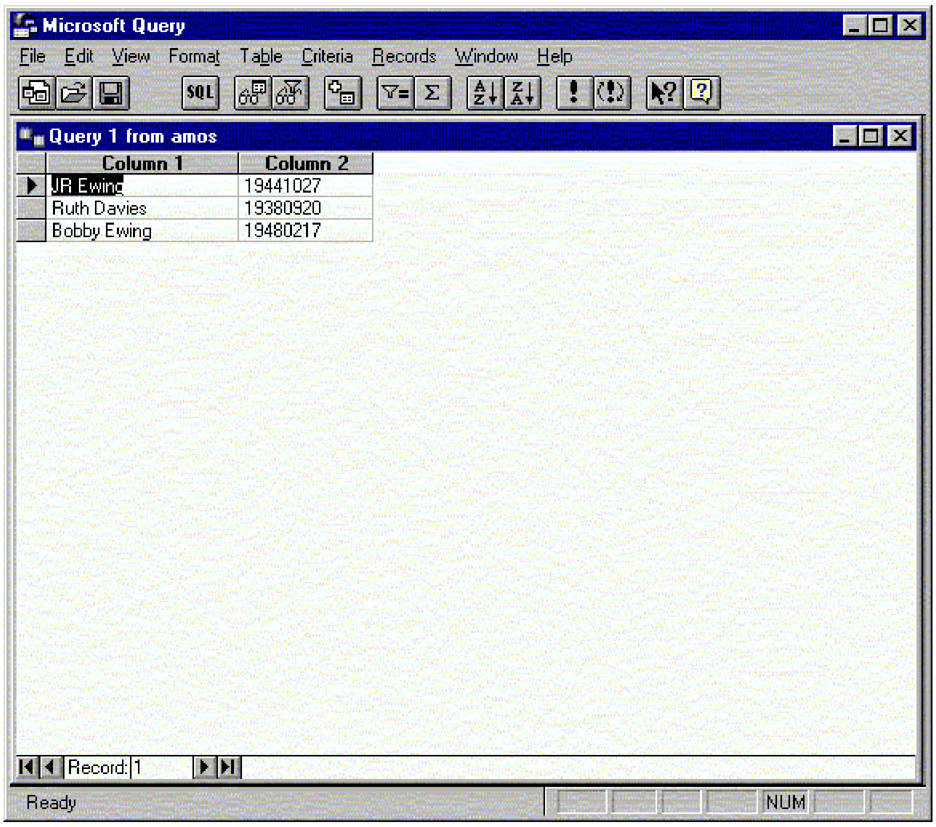
The server (foo) has a different database. At the client (the ODBC application), execute a query on the server:



```
amosql: at 'foo' select name(e), birthday(e) from employee e
where total_sales(e) > 483000;
```

Image 8-13. Entering a query to execute on the remote AMOS server.

Note that it is still possible to execute queries on the local database.
This example shows that the ODBC-driver really is a full-fledged AMOS system.



Column 1	Column 2
JR Ewing	19441027
Ruth Davies	19380920
Bobby Ewing	19480217

Image 8-14. Result of the remotely executed query.

9 Conclusion and future work

This chapter discusses the result of the work as well as possible improvements that can be made in the future.

9.1 Discussion

As has been shown in this report, the ODBC specification is rather loose and therefore it is possible to design a driver as different as the AMOS II ODBC-driver. It has also been shown that the implemented driver is not necessarily crippled by the fact that only simple *select* statements are allowed. The possibility to use AMOSQL as query language and the fact that the driver is in fact a complete AMOS system possibly makes the driver far more capable than other present drivers. However, updating the database using ODBC takes a lot of time simply because the entire database image has to be written to disk when committing. This is one of the disadvantages of main-memory databases, especially when used within an ODBC-driver. Another disadvantage of the chosen driver architecture is that it uses a lot of memory, at least compared to other drivers, and that a driver crash would also crash both the AMOS system and the application.

9.2 Future work

There is a lot of work that has to be done to make the driver a full implementation of the ODBC specification. The most important ones, as I see it are:

- To make it possible to connect to remote AMOS servers in a simple way and still use SQL as query language.
- To extend the parser and data model translation to handle more than *select* statements. Especially updating would be interesting to see, but also creation of new tables.
- To get rid of all bugs.

The first one should be rather simple to implement but the second would require a significantly more complex parser and a different way of mapping relational tables into objects. This would be a rather large work, perhaps suitable for a new master's thesis?

APPENDIX A: Flex specification

This appendix contains the Flex specification used to generate the lexical analyser.

```
/****************************************************
* Lexical analyser for the AMOS2 ODBC parser
* Marcus Eriksson, EDSLAB 1998
* Generate code by running: flex -P_odbc sql_lexer.l
****************************************************/
%{
#include "sql_parser.tab.h"
#include "parseutils.h"
#include <string.h>
#include <malloc.h>

#undef _odbcwrap
#undef YY_INPUT
#define YY_INPUT(b, r, ms) (r = my_yyinput(b, ms))

int lineno = 1;
void _odbcerror(char *s);

#define SV save_str(yytext);
#define TOK(name) {SV; return name;}
%}

%s SQL

%%

sql:{BEGIN SQL; start_save();}

/* literal keyword tokens */

<SQL>ANDTOK(AND)
<SQL>MINTOK(AMMSC)
<SQL>MAXTOK(AMMSC)
<SQL>SUMTOK(AMMSC)
<SQL>CHAR(ACTER)?TOK(CHARACTER)
<SQL>DISTINCTTOK(DISTINCT)
<SQL>FROMTOK(FROM)
<SQL>GROUPTOK(GROUP)
<SQL>INT(EGER)? TOK(INTEGER)
```

```
<SQL>NOTTOK(NOT)
<SQL>NULLTOK(NULLX)
<SQL>ORTOK(OR)
<SQL>ORDERTOK(ORDER)
<SQL>SELECTTOK(SELECT)
<SQL>UNIONTOK(UNION)
<SQL>WHERETOK(WHERE)
<SQL>HAVING     TOK(HAVING)
<SQL>ALL        TOK(ALL)

/* comparison */

<SQL>”=”TOK(EQ)
<SQL>”<>”{return(NEQ);}/* != in AMOS */
<SQL>”<“TOK(LT)
<SQL>”>”TOK(GT)
<SQL>”<=”TOK(LTE)
<SQL>”>=”TOK(GTE)

/* Reference to the “OID-column” */

<SQL>OID{return OID;}

/* Parameter marker */

<SQL>”?”{return PARAM;}

<SQL>[-+*/:();,]TOK(yytext[0])
<SQL>”.”{return yytext[0];}

/* names */

<SQL>[a-zA-Z][A-Za-z0-9_]*{return NAME;}

/* numbers */

<SQL>[0-9]+|
<SQL>[0-9]+”.”[0-9]* |
<SQL>”.”[0-9]*TOK(INTNUM)

<SQL>[0-9]+[eE][+-]?[0-9]+|
<SQL>[0-9]+”.”[0-9]*[eE][+-]?[0-9]+|
<SQL>”.”[0-9]*[eE][+-]?[0-9]+TOK(APPROXNUM)

/* strings */

<SQL>’[^’\n]*’{
  int c = input();
  unput(c);/* just peeking */
  if(c != ‘\”’)
  {
    SV;
    return STRING;
  }
  else
    yymore();
}

<SQL>'[^'\n]*${_odbcerror("Unterminated string");}
/* whitespace */

<SQL>\n{save_str(" "); lineno++;}

<SQL>[ \t\r]+{save_str(" ");}/* whitespace */

/* anything else */
.{_odbcerror("syntax error, invalid character");}

%%

extern char *myinputptr;  /* current position in buffer */
extern int mybufsize;     /* size of data */

int min(int a, int b)
{
  if (a < b)
    return a;
  else
    return b;
}

int my_yyinput(char *buf, int max_size)
{
  int n = min(max_size, mybufsize);

  if(n > 0)
  {
    memcpy(buf, myinputptr, n);
    myinputptr += n;
    mybufsize -=n;
  }
  else
    *buf=NULL;

  /*fprintf(stderr, "my_yyinput, copied %i characters. buf: %s\n\n", n, buf);*/
  return n;
}

void _odbcerror(char *s)
{
  extern char *parse_err;   /* String containing error message */
  sprintf(parse_err, "%s at %s\n", s, _odbctext);
}

int _odbcwrap()
{
  return 1;
}
```

```
un_sql()
{
  BEGIN INITIAL;
}
```

APPENDIX B: Bison grammar description

Bison grammar description used to produce a parsing function

```
/*************************************************************
 * Bison parser for the AMOS2 ODBC-driver
 * Parsing SQL statements
 * Marcus Eriksson, EDSLAB 1998
 * Generate code by running: bison -p _odbc sql_parser.y
 *************************************************************/
%{
   #include “parseutils.h”
   #include <string.h>

   #ifndef TRUE
     #define TRUE  1
     #define FALSE 0
   #endif

   extern char *_odbctext;      /* Defined in the lexer */
   extern unsigned num_params;  /* Defined by the caller. The number of parameters. */
   char rangevar[256], buf[256], tabvar[256];
   unsigned short from_seen=FALSE, table_seen=FALSE;
   /* Parameters are named ?1, ?2, etc. */

%}

%pure_parser  /* Reentrant parser */

%union
{
  char *strval;
}

%token NAME
%token STRING
%token INTNUM APPROXNUM

/* operators */

%left OR
%left AND
%left NOT
```

```
/* comparison */
%left EQ
%left NEQ
%left LT
%left GT
%left LTE
%left GTE

%left ‘+’ ‘-’
%left ‘*’ ‘/’
%nonassoc UMINUS

/* literal keyword tokens */

%token AMMSC AS BY LIKE ESCAPE EXISTS IS PARAMETER INDICATOR ALL
%token CHARACTER COMMIT DISTINCT DOUBLE FLOAT FROM GROUP USER HAVING
%token INTEGER NULLX ORDER REAL ROLLBACK SELECT TABLE UNION WHERE
%token OID PARAM

%%

/*****************************************************************/

/* RULES */

sql:select_statement ‘;’{end_sql();}
;

select_statement:SELECT opt_all_distinct selection table_exp
;

opt_all_distinct:/* empty */
/* |  ALL (not supported) */
|  DISTINCT
;

table_exp:from_clause opt_where_clause opt_group_by_clause opt_having_clause
;

from_clause:FROM table_ref_commalist{from_seen=TRUE;}
;

table_ref_commalist:table_ref
|table_ref_commalist ‘,’ table_ref
;

table_ref:table
|table range_variable
    {sprintf(buf, “%s %s”, tabvar, _odbctext);
    save_str(buf);}
;

range_variable:NAME
```

```
{strcpy(rangevar, _odbctext);
 if(!from_seen)
   {
     /* printf("%s\n", _odbctext); */
   }

}
;

opt_where_clause:/* empty */
|  WHERE search_condition
;

opt_group_by_clause:/* empty */
|  GROUP BY column_ref_commalist  {_odbcerror("GROUP BY not supported");}
;

column_ref_commalist:column_ref
|column_ref_commalist ',' column_ref
;

opt_having_clause:/* empty */
|  HAVING search_condition{_odbcerror("HAVING not supported");}
;

selection:scalar_exp_commalist
|'*'
;

scalar_exp:scalar_exp '+' scalar_exp
|   scalar_exp '-' scalar_exp
|   scalar_exp '*' scalar_exp
|   scalar_exp '/' scalar_exp
|   atom
|   column_ref
|   '(' scalar_exp ')'
|   function_ref
|   PARAM{sprintf(buf, "?%i", ++num_params);
 save_str(buf);}
;

scalar_exp_commalist:scalar_exp
|scalar_exp_commalist ',' scalar_exp
;

function_ref:AMMSC '(' scalar_exp ')'
;

atom:literal
;

column_ref:range_variable '.' NAME
       {sprintf(buf, "%s(%s)", _odbctext, rangevar);
        save_str(buf);
if(!from_seen)
  /* printf(“%s\n”, _odbctext); */
       }

| range_variable ‘.’ OID
{sprintf(buf, “%s”, rangevar);
 save_str(buf);
 if(!from_seen)
   /* printf(“OID\n”); */
}
;


/* search conditions */

search_condition:
search_condition AND search_condition
| search_condition OR search_condition
| NOT search_condition
| ‘(‘ search_condition ‘)’
| predicate
;


predicate:comparison_predicate
| existence_test
;

comparison_predicate:scalar_exp comparison scalar_exp
|scalar_exp comparison subquery
;

existence_test:EXISTS subquery{_odbcerror(“EXISTS not supported”);}
;

subquery:‘(‘ SELECT opt_all_distinct selection table_exp ‘)’
;

literal:STRING
|INTNUM
|APPROXNUM
;

table:NAME
{strcpy(tabvar, _odbctext);
 table_seen=TRUE;
 /* printf(“%s\n”, _odbctext); */
}
;

comparison:EQ
|NEQ{save_str(“!=”);}
|LT
|GT
```

```
|LTE
|GTE
;
```

APPENDIX C: Stored data and help functions

This generates the stored data and functions used by the driver.

```
/* Functions and tables stored in the database image. */
/* Used by the ODBC-driver. */

lisp;

;; Has been included in the standard AMOS image.
;; Return all supertypes above tp
;;(foreign-lispfn allsupertypes ((type tp)) ((type))
;;   (dolist (x (type-allsupertypes tp))
;;     (foreign-result x)))

;; Return all attributes for a type.
;; Don't return attributes defined in the type object (inherited attributes)
(create-function user_attributes ((type t)) ((function f))
   as (f)
   foreach ((type tp))
   where (and (= tp (allsupertypes t))
            (!= tp (typenamed "object"))
            (= f (attributes tp))))


;; create a new subtree in the type hierarchy.
;;(createtype 'ODBC "(object)")

;; create type to hold data for SQLGetTypeInfo (a subtype of ODBC)
(createtype 'odbc_type_info "(ODBC)")

;; create type to hold information for SQLTables.
(createtype 'odbc_tables_info "(ODBC)")

;; type to hold information for SQLColumns
(createtype 'odbc_column_info "(ODBC)")

;; type to hold information for the oid column in SQLColumns
(createtype 'odbc_oid_column "(ODBC)")

:osql
/*****************************************************************/
create function type_name(odbc_type_info)->charstring as stored;
create function data_type(odbc_type_info)->integer as stored;
create function column_size(odbc_type_info)->integer as stored;
create function literal_prefix(odbc_type_info)->charstring as stored;
create function literal_suffix(odbc_type_info)->charstring as stored;
create function create_params(odbc_type_info)->charstring as stored;
create function nullable(odbc_type_info)->integer as stored;
create function case_sensitive(odbc_type_info)->integer as stored;
create function searchable(odbc_type_info)->integer as stored;
create function unsigned_attribute(odbc_type_info)->integer as stored;
create function fixed_prec_scale(odbc_type_info)->integer as stored;
create function auto_unique_value(odbc_type_info)->integer as stored;
create function local_type_name(odbc_type_info)->charstring as stored;
create function minimum_scale(odbc_type_info)->integer as stored;
create function maximum_scale(odbc_type_info)->integer as stored;
create function sql_data_type(odbc_type_info)->integer as stored;
create function sql_datetime_sub(odbc_type_info)->integer as stored;
create function num_prec_radix(odbc_type_info)->integer as stored;
create function interval_precision(odbc_type_info)->integer as stored;

/* store info for SQLGetTypeInfo */

create odbc_type_info(type_name,data_type,column_size,literal_prefix,
literal_suffix,create_params,nullable,case_sensitive,searchable,
unsigned_attribute,fixed_prec_scale,auto_unique_value,local_type_name,
minimum_scale,maximum_scale,sql_data_type,sql_datetime_sub,num_prec_radix,
interval_precision) instances

/*
"real",SQL_DOUBLE,15,NULL,NULL,NULL,SQL_NO_NULLS,SQL_FALSE,
SQL_PRED_BASIC,SQL_FALSE,SQL_FALSE,NULL,
"double",NULL,4,SQL_DOUBLE,NULL,NULL,NULL) */

:real ("real",8,15,0,0,0,0,0,2,0,0,0,"real",0,4,8,0,0,0),

/* "integer",SQL_INTEGER,10,NULL,NULL,NULL,SQL_NULLABLE,SQL_FALSE,
SQL_PRED_BASIC,SQL_FALSE,SQL_FALSE,NULL,
"integer",NULL,NULL,SQL_INTEGER,NULL,NULL,NULL */

:int ("integer",4,10,0,0,0,0,0,2,0,0,0,"integer",0,0,4,0,0,0),

:oid ("oidtype",4,10,0,0,0,0,0,2,0,0,0,"oidtype",0,0,4,0,0,0),

/* "charstring",SQL_VARCHAR,255,"”","”","maxlength",SQL_NULLABLE,SQL_FALSE,
SQL_PRED_BASIC,NULL,NULL,NULL,"charstring",NULL,NULL,SQL_VARCHAR,
NULL,NULL,NULL) */

:string ("charstring",12,256,"”","”","maxlength",0,0,2,0,0,0,"charstring",0,0,12,0,0,0),

/* Return OID as an integer */
:userobj ("integer",4,10,0,0,0,0,0,2,0,0,0,"usertypeobject",0,0,4,0,0,0);

/* Functions for accessing TypeInfo */

/* All types */
```

```
create function ODBCGetTypeInfoAllTypes() -> <charstring,integer,integer,charstring,
charstring,charstring,integer,integer,integer,integer,integer,integer,charstring,integer,
integer,integer,integer,integer,integer> as
select type_name(t),data_type(t),column_size(t),literal_prefix(t),literal_suffix(t),
create_params(t),nullable(t),case_sensitive(t),searchable(t),unsigned_attribute(t),
fixed_prec_scale(t),auto_unique_value(t),local_type_name(t),minimum_scale(t),
maximum_scale(t),sql_data_type(t),sql_datetime_sub(t),num_prec_radix(t),
interval_precision(t)
from odbc_type_info t;

/* A specific type */
create function ODBCGetTypeInfoSpecificType(integer typ) -> <charstring,integer,integer,
charstring,charstring,charstring,integer,integer,integer,integer,integer,integer,
charstring,integer,integer,integer,integer,integer,integer> as
select type_name(t),data_type(t),column_size(t),literal_prefix(t),literal_suffix(t),
create_params(t),nullable(t),case_sensitive(t),searchable(t),unsigned_attribute(t),
fixed_prec_scale(t),auto_unique_value(t),local_type_name(t),minimum_scale(t),
maximum_scale(t),sql_data_type(t),sql_datetime_sub(t),num_prec_radix(t),
interval_precision(t)
from odbc_type_info t where data_type(t)=typ;

/*****************************************************************/
create function table_cat(odbc_tables_info)->charstring as stored;
create function table_schem(odbc_tables_info)->charstring as stored;
create function table_type(odbc_tables_info)->charstring as stored;
create function remarks(odbc_tables_info)->charstring as stored;

/* store info for SQLTables */
create odbc_tables_info (table_cat, table_schem, table_type, remarks) instances :odbc_tables
("<none>", "<none>", "TABLE", "AMOS2 TYPE");

/* create a function to call from SQLTables
   Returns the following:
1: TABLE_CAT
2: TABLE_SCHEM
3: TABLE_NAME
4: TABLE_TYPE
5: REMARKS
*/
create function ODBCGetTables() -> <charstring, charstring, charstring, charstring, charstring> as
select table_cat(t), table_schem(t), name(p), table_type(t), remarks(t)
from odbc_tables_info t, type p
where name(typesof(p)) = "usertype";

/* Get info on the table named 'table_name'. */
create function ODBCGetSpecificTable(charstring table_name) ->
<charstring, charstring, charstring, charstring, charstring> as
select table_cat(t), table_schem(t), name(p), table_type(t), remarks(t)
from odbc_tables_info t, type p
where name(typesof(p)) = "usertype" and name(p) = table_name;

/*******************************************************************/

/* Get "column" results */
```

```
create function methods(charstring table) -> function as
select f from function f where f =
user_attributes(typenamed(table));

create function method_results(function f) -> <charstring, type> as
select distinct name(f), t from integer k, type t, integer m where
<k, t, m> = argrestypes(resolvents(f)) and k = 2;

create function map_types(type t) -> integer as
select data_type(x) from odbc_type_info x where
name(allsupertypes(t)) = local_type_name(x);

/* This function returns all columns with their types (SQL-types) for a specific table. */
create function table_methods(charstring table) -> <charstring, integer> as
select nm, map_types(t) from charstring nm, type t where <nm, t> =
method_results(methods(table));

/*********************************************************************/

create function table_cat(odbc_column_info)->charstring as stored;
create function table_schem(odbc_column_info)->charstring as stored;
create function data_type(odbc_column_info)->integer key as stored;
create function type_name(odbc_column_info)->charstring key as stored;
create function column_size(odbc_column_info)->integer as stored;
create function buffer_length(odbc_column_info)->integer as stored;
create function decimal_digits(odbc_column_info)->integer as stored;
create function num_prec_radix(odbc_column_info)->integer as stored;
create function nullable(odbc_column_info)->integer as stored;
create function remarks(odbc_column_info)->charstring as stored;
create function column_def(odbc_column_info)->charstring as stored;
create function sql_data_type(odbc_column_info)->integer as stored;
create function sql_datetime_sub(odbc_column_info)->integer as stored;
create function char_octet_length(odbc_column_info)->integer as stored;
create function is_nullable(odbc_column_info)->charstring as stored;

/* Store info for SQLColumns */
create odbc_column_info(table_cat, table_schem, data_type, type_name, column_size, buffer_length,
decimal_digits, num_prec_radix, nullable, remarks, column_def, sql_data_type,
sql_datetime_sub, char_octet_length, is_nullable) instances

/* integer */
:int_col ("<none>", "<none>", 4, "integer", 10, 4, 0, 10, 0, 'object property function', 'NULL', 4, 0, 0,
'NO'),

/* double */
:real_col ("<none>", "<none>", 8, "real", 15, 8, 0, 2, 0, 'object property function', 'NULL', 8, 0, 0, 'NO'),

/* charstring */
:char_col ("<none>", "<none>", 12, "charstring", 256, 256, 0, 0, 0, 'object property function', 'NULL',
12, 0, 256, 'NO');

/*********************************************************************/
```

```
/* This function gets all columns of a specific table and returns a set as specified
   in the ODBC standard. */
create function ODBCGetColumns(charstring table) -> <charstring, charstring, charstring, charstring,
integer, charstring, integer, integer, integer, integer, integer, charstring, charstring,
integer, integer, integer, integer, charstring> as
select table_cat(t), table_schem(t), table, nm, data_type(t), type_name(t),
column_size(t), buffer_length(t), decimal_digits(t), num_prec_radix(t), nullable(t),
remarks(t), column_def(t), sql_data_type(t), sql_datetime_sub(t), char_octet_length(t),
2, is_nullable(t)
from odbc_column_info t, charstring nm, integer typ where <nm, typ> = table_methods(table)
and data_type(t) = typ;

/*********************************************************************/

create function table_cat(odbc_oid_column)->charstring as stored;
create function table_schem(odbc_oid_column)->charstring as stored;
create function data_type(odbc_oid_column)->integer as stored;
create function type_name(odbc_oid_column)->charstring as stored;
create function column_size(odbc_oid_column)->integer as stored;
create function buffer_length(odbc_oid_column)->integer as stored;
create function decimal_digits(odbc_oid_column)->integer as stored;
create function num_prec_radix(odbc_oid_column)->integer as stored;
create function nullable(odbc_oid_column)->integer as stored;
create function remarks(odbc_oid_column)->charstring as stored;
create function column_def(odbc_oid_column)->charstring as stored;
create function sql_data_type(odbc_oid_column)->integer as stored;
create function sql_datetime_sub(odbc_oid_column)->integer as stored;
create function char_octet_length(odbc_oid_column)->integer as stored;
create function is_nullable(odbc_oid_column)->charstring as stored;
create function column_name(odbc_oid_column)->charstring as stored;
create function ordinal_position(odbc_oid_column)->integer as stored;

create odbc_oid_column(table_cat, table_schem, column_name, data_type, type_name,
column_size,buffer_length, decimal_digits, num_prec_radix, nullable, remarks, column_def,
sql_data_type, sql_datetime_sub, char_octet_length, ordinal_position, is_nullable)
instances

/* Return OID as an integer */
:oid_col ("<none>", "<none>", "OID", 4, "integer", 10, 4, 0, 10, 0, 'object identifier', 'NULL', 4, 0, 0, 2,
'NO');

/* Get the OID column for a table. This column doesn't really exist. */
create function ODBCGetOIDColumn(charstring table) -> <charstring, charstring, charstring,
charstring, integer,
charstring, integer, integer, integer, integer, integer, charstring, charstring, integer, integer,
integer, integer, charstring> as
select table_cat(t), table_schem(t), tname, column_name(t), data_type(t), type_name(t), column_size(t),
buffer_length(t), decimal_digits(t), num_prec_radix(t), nullable(t), remarks(t), column_def(t),
sql_data_type(t), sql_datetime_sub(t), char_octet_length(t), ordinal_position(t), is_nullable(t)
from charstring tname, odbc_oid_column t where tname = table;
```

APPENDIX D: Implemented ODBC functions

In this appendix all implemented ODBC functions are presented. Some functions are more interesting than others, those functions are described in more detail.

All functions return a flag indicating if the call was successful or not, and if there are more information about the outcome of the call. The functions post error records which the application can examine to figure out what went wrong. A buffer might for example be to small and the driver might have truncated the data it returned in that buffer.
The functions are ordered by the order they were implemented, functions logically “belonging” together in sequence. First functions for allocating handles, freeing handles, connecting and disconnecting then executing statements, fetching results and so on.

SQLAllocHandle

Allocates memory for an environment, connection or statement and returns a handle to the allocated memory. How these “objects” are implemented is driver specific.
For example, this is an environment in the AMOS2 ODBC-driver:

```
/* Environment information */
typedef struct tagENV
{
   CRITICAL_SECTION lock; /* Protection from multiple threads */
   LIST diagnostics; /* Diagnostics records for this environment */
   unsigned num_connections; /* The number of AMOS connections */
   a_connection connection; /* Connection to AMOS */
} ENV;
```

Since AMOS is completely integrated into the driver in this case, the connection is actually in the environment instead of in the connection. Also, only one connection to the data source is allowed. The application can examine driver capabilities to find out if the driver supports multiple connections or not, but some application doesn’t respect this (all Microsoft Office applications). This means that the driver has to be able to fool the application that multiple connections are possible when in fact they are not.
As can be seen for a statement below, a statement “belongs to” a certain connection. (A connection can have multiple active statements.)

```
/* Statement information */
typedef struct tagSTMT
{
  CRITICAL_SECTION lock;
  DBC *dbc; /* □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ */
  DESC *ARD, *APD, *IRD, *IPD, *imp_ARD, *imp_APD, *imp_IRD, *imp_IPD;
  LIST diagnostics;
  a_scan scan;
  a_scan secondary_scan;
  a_scan special_scan;
  a_tuple tuple;
  a_tuple argl;
  oidtype fun;
  SQLUINTEGER current_row;
  SQLUINTEGER max_length;
  enum QUERY_TYPE query_type;
  char *last_prepared;
} STMT;
```

SQLFreeHandle

Free a previously allocated environment, connection or statement.

SQLDriverConnect

Takes a connection handle and a connection string (with user name, password and other necessary information) and connects to a data source. If necessary, that is, if the connection string is incomplete, pop up a dialogue box asking for more information.

SQLDisconnect

Takes a connection handle and disconnects from the associated data source.

SQLFreeStmt

Takes a statement handle as parameter and either frees all bound variables or all bound columns to that statement depending on a second parameter.

SQLSetEnvAttr

The application can set various environment attributes for the driver, but this is not supported in this case.

SQLSetConnectAttr

Same as above, but for connection attributes. The only one possible to change is auto-commit, by turning it on or off.

SQLSetStmtAttr

Same as above but for statement attributes. Some attributes can be set, but they are not very interesting.

SQLGetEnvAttr

Used for getting environment attributes, for example if strings are null terminated or not.

SQLGetConnectAttr

Used for getting connection attributes, for example if autocommit is on or off.

SQLGetStmtAttr

Used for getting statement attributes.

SQLGetTypeInfo

Used for getting information about supported data types. Calls a stored AMOS function. (See chapter on stored functions above.) The result is retrieved exactly as if an ordinary query was sent to the data source, that is, by subsequent calls to SQLFetch.

SQLTables

Get information on all available tables. (See chapter on stored functions above.)

SQLColumns

Get all columns in a specific table. (See chapter on stored functions above.)

SQLGetDescField

Get a certain field from a descriptor record.

SQLGetDescRec

Get a descriptor record.

SQLGetInfo

Get info about driver capabilities, among other things. For example what datatypes a driver can convert.

SQLGetDiagField

Get a certain field from a diagnostics record. The driver posts diagnostic records after every application call to a driver function. The diagnostic record contains information on if the execution was successful or not. For example, the driver might post a message text for the user to read.

SQLGetDiagRec

Get a certain diagnostics record. The driver might post multiple records, which are numbered.

SQLPrepare

Prepare a statement for later execution. The SQL statement is translated to AMOSQL and then a transient function is created (which is lost when disconnected from AMOS). Statements already in AMOSQL are not allowed. Use direct execution for this.

SQLExecDirect

Direct execution of a statement. The SQL statement is translated to AMOSQL using the Bison generated parser. No parameters are allowed for direct execution.

SQLExecute

Execute a previously prepared function.

SQLBindParameter

Bind all parameters for execution, that is, supply values for all parameters. Called prior to SQLExecute. After execution, new parameters may be bound.

SQLEndTran

End the current transaction. A flag, SQL_COMMIT or SQL_ROLLBACK is supplied. When committing, the work is committed and since AMOS is a main-memory database, the database image is also saved to disk. Therefore this is a very slow way of committing the work, so make sure autocommit is off, otherwise the database will be saved after every executed statement.

SQLCancel

Cancel the current statement. Actually implemented as a call to SQLFreeStmt with the option SQL_CLOSE (Close cursor associated with statement and discard pending results).

SQLNumResultCols

The number of columns in the result set of an executed query.

SQLDescribeCol

Description of a column in the result set such as data type and column name.

SQLBindCol

Bind columns in the result set, that is tell the driver what columns in the result set you are interested in retrieving values from. It is perfectly OK to execute something like

```
SELECT * FROM <table>
```

and only bind one of the (possibly) many columns in the result. The driver automatically converts the result to the correct data type.

SQLFetch

Fetch the results of the previously executed statement into all bound columns.

SQLColAttribute

Get column attributes for a specific column in the result set. Approximately the same thing as SQLDescribeCol, only a few more options.

APPENDIX E: Example database

Database used in the examples:

```
create type named_object properties (name charstring);
create type person subtype of named_object;
create type dept subtype of named_object properties (location charstring);
create type teacher subtype of person properties (works_at dept, email charstring);
create type student subtype of person properties (major charstring, email charstring);
create type subject subtype of named_object properties (teaching_dept dept);

create dept(name, location) instances
:ida(’IDA’, ’E-house’),
:ifm(’IFM’, ’F-house’),
:mai(’MAI’, ’B-house’);

create teacher(name, works_at, email) instances
:tore(’Tore Risch’, :ida, ’torri@ida.liu.se’),
:peter(’Peter Fritzon’, :ida, ’petfr@ida.liu.se’),
:tommy(’Tommy Olsson’, :ida, ’tao@ida.liu.se’),
:jesper(’Jesper Andersson’, :ida, ’jesan@ida.liu.se’),
:jan(’Jan Lundgren’, :ifm, ’jlu@ifm.liu.se’),
:bengt(’Bengt Josefsson’, :mai, ’bejos@mai.liu.se’);

create student(name, major, email) instances
:macke(’Marcus Eriksson’, ’CS’, ’x98marer@ida.liu.se’),
:peps(’Per Persson’, ’Physics’, ’perpe@ifm.liu.se’),
:johan(’Johan Sandberg’, ’Physics’, ’johsa@ifm.liu.se’),
:sussie(’Susanne Andersson’, ’Mathematics’, ’suand@mai.liu.se’),
:tony(’Tony Bengtsson’, ’Mathematics’, ’tony.bengtsson@positionett.se’),
:sara('Sara Hjärne', 'Physics', 'sarhj302@student.liu.se');

create subject(name, teaching_dept) instances
:cs('CS', :ida),
:ph('Physics', :ifm),
:ma('Mathematics', :mai);

create type pet subtype of named_object;
create function length(pet) -> real;

create pet(name, length) instances :robban('Robin', 34.5), :hubbe('Hubert', 10.8);
```

References

[1] K North, *Windows Multi-DBMS Programming*, Wiley 1995

[2] T Risch, *AMOS2 External Interfaces*, Dept. of Computer and Information Science, Linköping University 1998

[3] *Microsoft ODBC 3.0 Programmer's Reference and SDK Guide*, Microsoft Press 1997

[4] R Signore, J Creamer, M. O. Stegman, *The ODBC Solution: open database connectivity in distributed environments*, McGraw-Hill 1994

[5] S Flodin, T Risch, M Sköld, M Werner, *AMOS2 User's Guide*, Dept. of Computer and Information Science, Linköping University 1998

[6] C. J. Date, *An introduction to database systems*, Addison-Wesley 1995

[7] J. R. Levine, T Mason, D Brown, *lex & yacc*, O'Reilly 1990

[8] M Sköld, *Active Rules based on Object Relational Queries*, Dept. of Computer and Information Science, Linköping University 1994

[9] W Litwin, T Risch, *Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates*, IEEE Transactions on Knowledge and Data Engineering Vol. 4, No. 6, December 1992

[10] D Fishman et al, *Overview of the Iris DBMS*, Object-Oriented Concepts, Databases, and Applications, ACM press, Addison-Wesley 1989

[11] G Wiederhold, *Mediators in the Architecture of Future Information Systems*, IEEE Computer, March 1992

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

LINKÖPINGS UNIVERSITET

Avdelning, Institution
Division, department

Institutionen för datavetenskap

Department of Computer and Information Science

Datum
Date

1999-05-06

Språk
Language

☐ Svenska/Swedish
[✓ Engelska/English]
☐ ____________

Rapporttyp
Report category

☐ Licentiatavhandling
[✓ Examensarbete]
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ ____________

ISBN

ISRN

Serietitel och serienummer
Title of series, numbering

ISSN

LiTH-IDA-Ex- 99/50

URL för elektronisk version

Titel
Title

En ODBC-driver för medlardatabasen AMOS II.
An ODBC-driver for the mediator database AMOS II.

Författare
Author

Marcus Eriksson

Sammanfattning
Abstract

ODBC (*Open Database Connectivity*) is a standardized application programming interface developed by Microsoft. By using the ODBC interface, applications can access a wide variety of data sources using the same source code. Prior to ODBC, applications written to access data stored in a Database Management System (*DBMS*) had to use the proprietary interface specific to that database. If application developers wanted to provide their users with access to data in more than one data source, they had to code to the interface of each data source. Naturally, applications written in this manner are difficult to code, difficult to maintain, and difficult to extend. The ODBC architecture was designed to permit maximum interoperability. It allows application developers to create an application without targeting a specific DBMS. End users can then use the application with the DBMS that contains their data by adding modules called database *drivers*, which are dynamic-link libraries (*DLLs*).
Today, ODBC has become the industry standard for interoperability with relational databases. Database management systems with ODBC-drivers can interoperate with hundreds of different applications.

AMOS II is a light-weight, main-memory, object-relational database kernel, running on the Windows NT platform. It contains a relationally complete query-language, AMOSQL. The purpose of this work is to develop an ODBC-driver for AMOS II.

Nyckelord
Keywords

ODBC, database, AMOS, object-relational

95-10-25/lisli
IdaMallar