Uppsala Master's Theses in
Computing Science 274
Examensarbete MN3
2004-07-01
ISSN 1100–1836

PHP Integration with object relational DBMS

Christian Werner

Information Technology
Computing Science Department
Uppsala University
Box 337
S-751 05 Uppsala
Sweden

Abstract

This report describes how to extend a functional mediator system Amos II for allowing access from web servers through PHP. Several possibilities are analysed to combine the Amos II external interface with PHP. Based on this discussions, new functionality has been added to the PHP language by implementing a PHP external module. A basic API between PHP and Amos II is proposed in this workout. The interface was illustrated by implementing a web interface to a simple database. Further studies and experiences from this illustration resulted in a simplified and more dynamic interface definition based on PHP arrays.

Supervisor: Tore Risch
Examiner: Tore Risch

Passed:

Contents

1 Introduction

The definition and properties of an API between Amos II and PHP are the main topics of this report. Amos II (Active Mediator Object System) is an extensible functional multi-database system [1]. Functional queries, expressed in AmosQL, can be executed by one database or over a federation of distributed databases. AmosQL is a query language similar to the OO parts of SQL-99. Furthermore Amos II can be set up as a stand-alone main-memory object-relational DBMS.

Two kinds of interfaces are offered between Amos II and the programming languages C [5] or Java [6], the *callin* interface and the *callout* interface. To call Amos II from an application the *callin*-interface can be used. Establishing connections, sending queries and dealing with result sets are tasks of this interface. Contrary the *callout*-interface is able to call external functions from C or Java. Once known to Amos II , these external functions can be used in AmosQL queries. The callin and callout interfaces permit the development of *wrappers*. Wrappers are small program modules that can access external data, for example XML-documents, relational databases, web forms, MIDI files and web browsers.

PHP (PHP: Hypertext Preprocessor) is a server side scripting language [8]. On the one hand PHP scripts can be executed on the command line. This is very helpful feature for debugging. On the other hand PHP is able to run inside a PHP-enabled web server. In this context PHP can be used to create interactive web pages. The core of PHP, *Zend*, is extensible which offers the possibility to access external systems like Amos II in this project. There are already existing several PHP extensions to various relational DBMS such as Oracle, MySQL, DB2 and ODBC. The goal of this project is the implemention of new PHP functions that call Amos II systems.

This documentation deals with designing and implementing an API between Amos II and PHP. First of all, interfaces to Amos II and PHP in a web server are analyzed in order to combine these two programs. Based on this analysis and previous Amos II interfaces for C and Java, the basic API was developed.

Several features have been considered while implementing the extension. Now it is possible to establish either a tight connection or a client/server connection. With a tight connection the Amos II database runs inside the PHP engine as a subsystem. The other possibility is to run Amos II as a seperate server. PHP scripts can send their requests to this server where they are worked on and answered.
Another very interesting feature is the passing of Amos II object identifiers, *OID*s, between Amos II systems and PHP. Thus Amos II objects can easily be accessed from PHP.
The error management of Amos II is integrated with the error management of PHP.

The defined API is illustrated by the implementation of a web interface to a simple database. An addressbook containing person and address objects is stored in this database. The web interface tests the implemented functions. Based on experiences of using the simple interface and further studies of PHP a simplified and dynamic interface based on PHP arrays is proposed.

This report is constructed as follows. First of all the Amos II system is detailly described in chapter 2. The architecture behind Amos II, the data model based on Amos II and the interfaces to this system are explained. Section 3 explains the basic structure of PHP and the possibility to extend PHP's core, Zend. It is furthermore investigated how to connect PHP types with the ones from Amos II, how to pass OIDs between the two systems and the integration of the two error managements. A very important point for server systems, garbage collection, is also investigated in this chapter. After this, performance observations are presented in chapter 4. Comparisons to a similar language, JSP, have additionally been made in this chapter. A complete function reference is given in chapter 5. To test the system an example PHP program has been implemented and is described in chapter 6. With progressive studies a very important improvement is proposed in section 7, the combination of Amos II tuples and PHP arrays. Finally chapter 8 gives a conclusion and some ideas for possible future works.

2 The Amos II system

Amos II is a distributed mediator system with an object oriented and functional data model. The name Amos stands for *Active Mediator Object System.* Queries to that data model are written in AmosQL, a relationally complete functional query language. The system can consist of several autonomous and distributed Amos II peers. These peers can interoperate through its distributed multi-database facilities. Each mediator peer offers three possibilities to access data:

- Access to data stored in an Amos II database.
- Access to wrapped datasources.
- Access to data that is reconciled from other mediator peers.

Especially the second point makes Amos II extensible. New application oriented data types or operators can easily be wrapped. And it is possible to add them to the query language, AmosQL. Thus a powerful query and data integration is offered by the Amos II system.

First of all this chapter deals with the mediator-wrapper approach, on which the Amos II system is based. To put this into concrete terms, the Amos II architecture is described after it. As the main purpose of Amos II is using it as a database, the data model of Amos II is described. Finally the *callin*-interface, basically used to extend PHP, is explained.

2.1 The mediator-wrapper architecture

Today a lot of applications, especially web applications, are developed that need to access databases. Therefore an increasing number of distributed databases is in use. Online travel agencies for example must be able to access databases of flight companies to gain information of empty seats, of airports to see arrival and departure timetables and of hotels to search for free rooms. Of course these information won't be stored in one database, but each flight company, airport and hotel will have their own database. Thus access to distributed databases is necessary.

Furthermore there exists a large amount of possibilities to store data. One company having this database and that schema, the next company might have a completely different view on the stored data. Thus an application needs the possibility to access herterogeneous data sources.

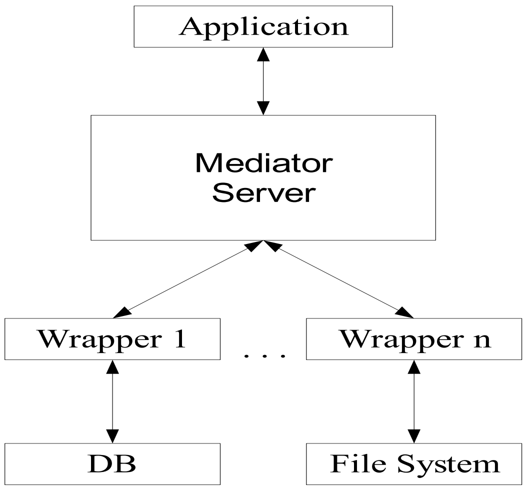

Figure 1: Mediator Wrapper Architecture

Therefore the mediator/wrapper approach, first proposed by Wiederhold [14], gives a good support for applications to access distributed and heterogeneous data sources. But what is this approach?

A mediator system consists of a mediator server and one or several wrappers. The *mediator server* is a central software module that comprises a *common data model* (CDM). This CDM shares its domain-specific knowledge about data with higher levels of mediator layers or with applications. The task of a mediator is to answer queries that are sent from applications to the mediator's common data model. The query will be split depending on the data and capabilities of the target data sources. As a mediator can regard other mediators as data sources, a network of mediators, consisting of several layers, can be created. In such a network only *primitive* mediators should have access to data sources. Then higher layers, consisting of an advanced data abstraction, can be accessed by applications.

One mediator can access different autonomous data sources, like databases, XML files or object stores, but never directly. Each data source can be accessed through software interfaces, called *wrapper*. Queries, sent from the mediator, are translated by the wrapper to a data source specific format and thus hides the heterogenity of that data source. After that the wrapper retrieves the query result, which has to be translated again into the common data model of the corresponding mediator. The translated data is passed to the mediator. There all results from all data sources are integrated and returned to the application. Figure 1 illustrates the way of a query beginning at the application, through the mediator server and the wrappers to the data sources. Then the result set takes the same way up.

2.2 The Amos II architecture

The Amos II system is based on the mediator/wrapper architecture. It is a distributed mediator system consisting of one or several mediator peers. These peers can communicate via the Internet using TCP/IP. Each peer offers its own virtual functional database layer, consisting of

- data abstractions that provide transparent functional views for clients and other mediator peers to access data sources,

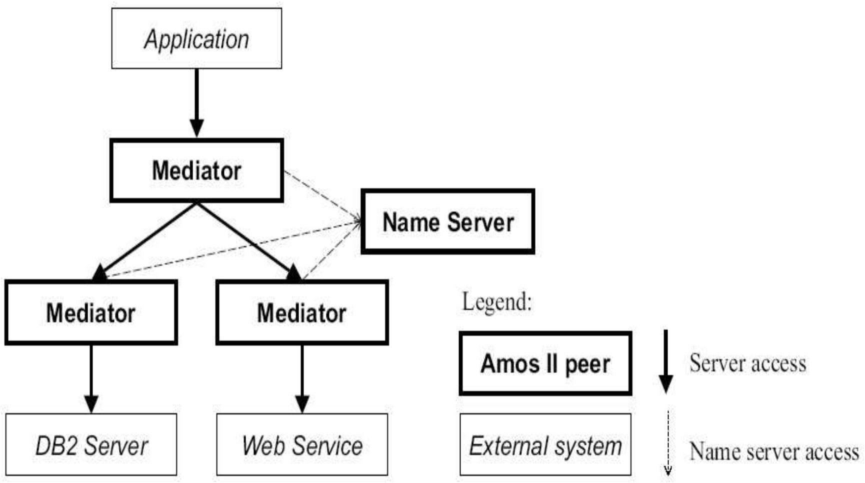

Figure 2: Example of an Amos II system with three mediators, [1]

- a functional query language, AmosQL,
- a storage manager
- a recovery manager and
- a transaction manager.

The core of Amos II is a open, light-weight and extensible database management system (DBMS).

In Amos II it is possible to build up layers of peers with a dynamic communication topology. A *distributed mediator query optimizer* optimizes this communication topology. To compute an optimized execution plan for a given query, data and schema information are exchanged between the peers. In figure 2, the high level mediator defines mediating functional view integrating data from them. The views include facilities for semantic reconciliation of data retrieved from the two lower mediators. The two lower mediators translate data from a wrapped relational database and a web server, respectively. They have knowledge of how to translate AmosQL queries to SQL through JDBC and, for the web server, to web service requests. The description of figure 2 can also be found in [1].

To summarize this, figure 2 illustrates the distribution of mediator peers. Two distributed data sources offer through three distributed mediator peers their data to an application. Communication between peers is illustrated by thick lines, where the arrow indicates the peer running as server. One special mediator peer is listed in the figure, the *name server*. Every mediator peer must belong to a group and every group of mediator peers must have a name server. The name server stores meta-information like

- names of peers,
- locations of peers and
- additional data

about all peers in that group. It must be mentioned that the mediator peers forming a group are still autonomous and there exists no control schema in the name server. And it is left to each peer to describe its own local data view and data sources. In what way is the name server involved in peer communication? The kind of communication is done through messages that can request or deliver data. To avoid a bottleneck the name server is only involved when a new mediator peer wants to register to the group. As soon as mediators are known to each other they can communicate directly without any information from the name server. In figure 2 the communication with the nameserver is illustrated by dotted lines. The name server always acts, with regard to the other mediator peers, as a server.

To access external data sources an Amos II mediator peer can have one or several *wrappers*. A wrapper is a small program module that translates queries received by the mediator and forwards them to the data source. It is also responsible to handle the result set returned from that data source. The data must be adapted to the mediator's data schema. Therefore the wrapper must contain information about the meta-data and the data itself of the data source. It furthermore must contain rewrite rules to translate the queries. As follows a number of tasks for a wrapper is listed ([1]):

- *Schema importation* translates schema information from the sources into a set of Amos II types and functions.
- *Query translation* translates internal calculus representations of AmosQL queries into equivalent API calls or query language expressions executable by the source.
- *Source statistics* computation estimates costs and selectivities for API calls or query expressions to a data source.
- *Proxy OID generation* executes in the source query expressions or API calls to construct *proxy OIDs* describing source data.
- *OID verification* executes in the source query expressions or API calls to verify the validity of involved proxy OIDs, in case they have become invalid between different query requests.

As soon as a wrapper is defined for a special data source, every query written in AmosQL can be rewritten.

To summarize and to put everything together, the route of a query is observed. Figure 3 gives a picture of such a route. An application sends a query to any mediator peer using AmosQL. In dependence of the data to access, the query will be split and forwarded to the wrappers. There the query is translated into a format that is analyzable by the data source. The right side of the picture shows the way of the result set. The data sources deliver their query result back to the wrapper that translates the data to the CDM of the mediator. There all result sets are unified and returned to the application.

The first time an Amos II server is initialized it runs initially as a stand-alone database in single-user mode. For this project it is also necessary to run Amos II as a server. Therefore the nameserver must first be started by using

```
nameserver(Charstring name)->Charstring
```

with *name* being the mediator name that is registered with starting the name server. If *name* is an empty string, only the name server is started. All other peers, that want to belong to the same group, must register to the name server of the current system by using

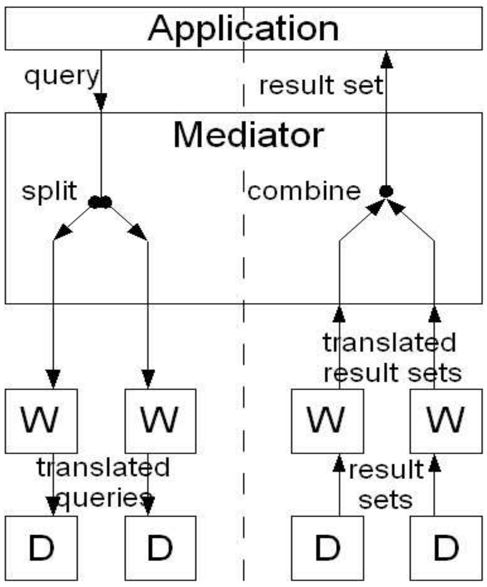

Figure 3: Example for a query execution

```
register(Charstring name)->Charstring .
```

For preparing the mediator to receive queries start the listening loop:

```
listen() .
```

2.3 Data model

The basic components of the data model of Amos II are *objects*, *types* and *functions*.

2.3.1 Types

For each entity type in an Entity-Relationship diagram an Amos II type is created. Types unite objects with similar properties. Furthermore multiple inheritance is offered, what means a supertype/subtype hierarchy can be built. An instance of a type is always an instance of all supertypes. If an object inherits from more than one type it gains all properties from all supertypes.

Two kinds of types are existing, *stored* and *derived* types. Derived types are mainly used for reconciliation, while the stored types are defined and stored in an Amos II database. The following command creates a person and an assistant type in Amos II:

```
create type Person;
create type Assistant under Person;
```

Every Amos II peer offers a basic type hierarchy that can be seen in figure 4. The root element is named Object. All (system and user) defined type names are stored in a type named *Type*. Again all functions (described in chapter 2.3.2) are instances of a type named *Function*. When a user defines a type it is always a subtype of type *Userobject*.

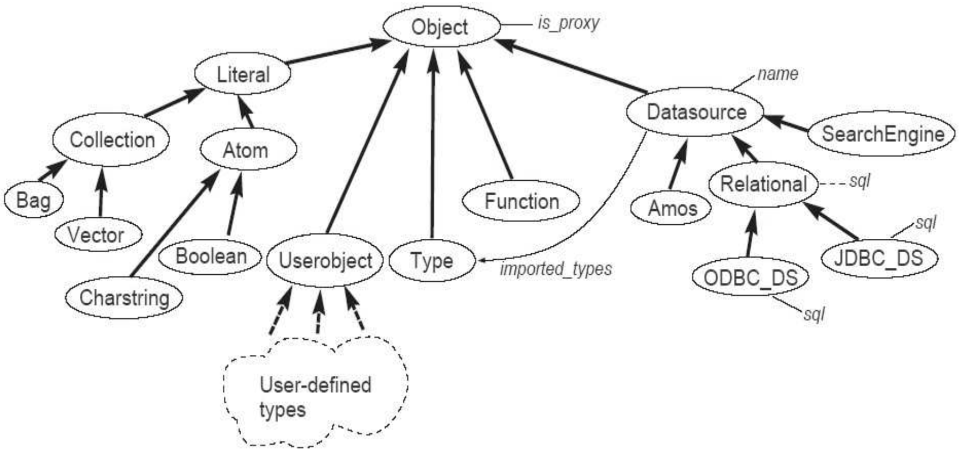

Figure 4: System type hierarchy

2.3.2 Functions

Functions model the relationship between objects, model properties of objects and computations over objects. In functional queries and views they are representing the basic primitives. As already mentioned, functions are instances of the type *Function*.

The function's *signature* contains information about all arguments, such as the types and optional names, and about the result of the function. The next example shows the signature of a function modeling the attribute `street` of type *Address*:

```
street(Address)->Charstring
```

2.3.3 Objects

Objects in Amos II are corresponding to Entities in an Entity-Relationship diagram and are instances of Amos II types. Everything in Amos II is represented as an object, independent whether the object is user-defined or system-defined. *Literals* and *surrogates* are are the main kinds for representing objects. Literal objects are primitive objects like integers, strings or even *collections* that represent arrays of other objects. In addition to this are surrogates that are created by the user or the system and are describing real world entities.

An *address*-object in Amos II can be created with a command as follows:

```
create address instances :hereiswhereilive;
```

When a query requests this object, the returned result will be displayed similar to this scheme:

```
#[OID 1101]
```

The system assigns unique object identifiers (OIDs) to all surrogate objects. In combination with the external interface for the programming language C these OIDs are stored as *object handles*. An object handle is a reference to any kind of data stored in Amos II databases such as numbers, strings, Amos II OIDs and other internal structures. Reversed no object handle that references a literal object has an OID.

It will be discussed later in chapter 3.4, how object handles are passed from C to PHP and the other way.

2.4 External interface

Applications that are developed in programming languages and want to access the Amos II system require special interfaces to the mediator layer. Interfaces to several programming languages are already existing:

1. **Lisp**: Amos II can be called from an extended Lisp language, named *ALisp*. The AmosQL parser is based on Lisp macros that translate queries and statements written in AmosQL to Lisp code. The interface from ALisp to Amos II is an *embedded query* interface. As explained in [5], this method for accessing Amos II can be very efficient.
2. **Java**: The most convenient way to write Amos II applications is using the Java interface that is described in [6]. The Java interface is divided into a *callin* and *callout* interface. The use of these two interfaces is similar to the ones from C. Thus the use is only described for C, but the technic is explained shortly.
   A driver program, called *JavaAMOS*, is necessary to call Amos II from Java code and is provided through *javaamos.jar*. For using the callout interface the *multi-directional foreign function* interface can be used.
3. **C**: The external interface to C is used for this project and is thus explained in detail below.

It is PHP that needs to be extended in this project. As the core of PHP, *Zend*, is written in C, the study of the C interface has been the basics for implementing the API of chapter 5. The rest of this chapter deals with this interface.

The external interface to C is divided into the *callin* and the *callout* interface.

- For calling Amos II from a C program the *callin* interface can be used. As it appears between the Amos II kernel (peer) and the PHP extension, *callin* makes the mediator-wrapper available to PHP.
- With the *callout* interface external procedures can be called from AmosQL. There is a similarity to *data blades* [18] in object-relational databases and foreign functions can be used as a form of *stored procedures*.

Having a more detailed look on the *internal* interface, there appear two ways to call Amos II from C:

- The evaluation of passed strings containing AmosQL statements is done by the *embedded query* interface. Some basic functions are provided to the programmer to handle results of AmosQL statements.
- Since the *embedded query* interface always has to parse and compile AmosQL statements it is rather slow. To handle this disadvantage, the *fast-path* interface is able to call predefined Amos II functions from C.

```
typedef struct {
   int status;
   a_connection con;
   char *session_id;
} amos_con;

typedef struct {
   int status;
   a_scan scan;
   int con_index;
} amos_scan;

typedef struct {
   int status;
   a_tuple tuple;
   int con_index;
} amos_tuple;
```

Figure 5: Data structures used in the PHP extension

In what way can Amos II be connected or how can result sets be handled in C code? There are two answers for the first question:

- On the one hand a *tight connection* to an Amos II system can be established. Here Amos II is directly linked with an application program in C and thus runs as an *embedded database* inside the application. As the same address space is used this is the fastest possibility between an application and Amos II. However only a single application can be linked to Amos II. Another disadvantage with the tight connection is that errors appearing in the application may also cause the crash of Amos II.
- Amos II can, as already described, act as an server. An application can establish a *client-server connection* to the Amos II peer. With this connection several applications can access the same Amos II concurrently. If one application crashes, it will have no influence on Amos II. Speed is a disadvantage for this kind of connection.

A *connection handle* to an Amos II database is stored in a C variable of type `a_connection`. The connection is established by calling

```
int a_connect(a_connection c, char *dbname, int catcherror);
```

with `dbname` being the name of the database peer to connect to. If `dbname` is an empty string, then a tight connection is established. Otherwise this string must be a name of an Amos II mediator peer known to the nameserver. In the PHP extension the connection handle is stored in an array of type `amos_con`. See figure 5 for more details. The element `status` indicates whether there is a valid connection handle `con` stored or not. Additionaly the string `session_id` is used for garbage collection (see section 3.6) and stores either the value `"0"` if connection `con` is established outside of a session, otherwise the session identifier is stored in this variable.

Now the connection is established and queries can be sent to an Amos II system that can return result sets. *Scan handle*s are holding the result set of queries and are returning them to the application as *scans*. Scans are specified by the type `a_scan` in Amos II and by `amos_scan` in the PHP extension. Again `status` indicates a valid scan. A scan handle is always created in correlation with a connection handle. This information is stored in `con_index`.

A scan is a stream of *tuples* that hold the actual values. Tuples are stored by using type `a_tuple` in Amos II and `amos_tuple` in PHP. The variables in the PHP structure are the same as in `amos_scan`.

3 PHP integration

This report deals with integrating Amos II into PHP. First of all a short introduction of the PHP environment in section 3.1 is done, followed by a description for extending PHP in chapter 3.2. The next topic to be studied is how to pass OIDs of Amos II objects to the user of PHP. In case of an error thrown in Amos II, the error managements of Amos II and PHP must be combined. Section 3.5 deals with this. Finally, but very important, techniques for garbage collection to cleaning used storage are described.

3.1 Introduction to PHP

PHP is above all a scripting language that can be embedded into HTML and is therefore mainly used as a server side scripting language. The shortcut PHP is a recursive acronym for '**P**HP: **H**ypertext **P**reprocessor'.

There are three possibilities for using PHP. It can be used

- as a server side scripting language,
- for command line scripting or
- for client-side GUI applications.

The first point is the most important one. As soon as a client (e.g. web browser) requests a PHP site from the server, PHP is called by the server to translate the PHP page into pure HTML code. The resulting HTML page is then returned to the client. The advantage of this architecture is that the PHP code from the PHP page cannot be seen by the user. The page is translated on the server and the user only sees the resulting HTML page. For this project the Apache Web Server is used, as Apache already has a PHP integration.

PHP also offers the possibility to execute scripts via the command line. This feature has been used in this project to test the implemented Amos II functions (described in section 5).

The third possibility to use PHP is developing GUI applications with the help of the PHP-GTK extension. For this work that feature is not of interest.

3.2 Extending PHP

For calling Amos II from PHP code, the functionality of PHP must be extended. But how to extend this language?

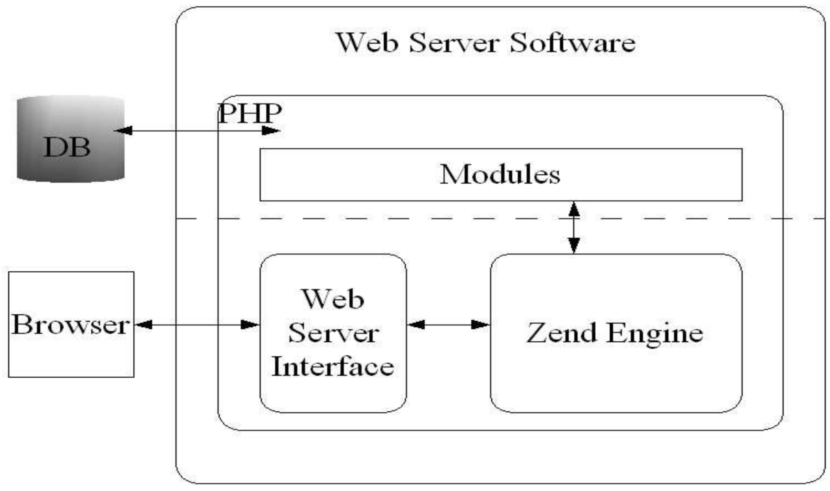

Figure 6: The structure of PHP

3.2.1 Internal PHP structure

First of all the structure of PHP must be discussed. As already mentioned, PHP is a web script interpreter. To implement such an interpreter, three parts are necessary:

- The *interpreter* that analyses the input code, compiles it and executes it.
- The *functionality* part offers function calls to the user.
- An *interface* to the webserver.

Therefore PHP is split up into two parts, the PHP language and the real core of PHP, named *Zend*. Zend contains the interpreter and offers in addition some basic functionality to the user via the *Zend API*. PHP takes the rest of the functionality and the interface part. Figure 6 illustrates the structure of PHP.

PHP can be extended at three points:

- External Modules
- Built-in Modules
- Zend Engine

External modules are loaded at script runtime by using the '`dl(modulename)`' command. A shared object is loaded from the disk and all functionality of the module is made available to the current script. At the end of the script the module is unloaded and discarded from memory.
An advantage of this procedure is, when changing the module only that module must be recompiled. PHP itself need not be recompiled and thus the size of PHP remains small and uses less memory. But as the module has to be loaded every time a script is being executed, PHP is getting quite slow. Furthermore external additional files clutter up the disk and every script that wants to call functions from an external module must include a call to `dl()`.

Thus external modules are only recommendable for small programs, or modules that are rarely used.

The second possibility, built-in modules, avoids the disadvantages of an external module by compiling it into PHP. Thus each time a PHP process is started, the functionality is automatically available to every running script, without calling the function `dl()`. The disadvantages are if changes to built-in modules are made recompiling of PHP is required and the PHP binary grows and consumes more memory.
Thus built-in modules can be used when it has a solid library of functions that remain relatively unchanged. The disadvantage recompiling PHP is quickly compensated by the speedup of the built-in module.

The last possibilty to extend PHP is to change the Zend Engine itself. This will result in a change of the PHP language behaviour. But as mentioned in [8] this precedure is not recommended as changes result in incompatibilities with the rest of the world. All changes to the PHP core will disappear with the next version of PHP, or have to rewritten.

This project uses an external module. In that way a recompilation of PHP can be avoided. As explained above the module must be loaded dynamically by the `dl()`-command, which will probably reduce the performance. It is also possible to tell PHP to load the module automatically by adding the Amos extension to file 'php.ini'. The last possibility is used in this project and no real performance problems have been observed. The next thing to discuss is how to write a PHP module.

3.2.2 PHP module structure

PHP offers a lot of macros to implement a module. The basic structure for the Amos II extension to PHP can be seen in figure 7.

First of all the header files for PHP and Amos II have to be included. The next important entry is a list of all implemented PHP functions. The list

```
function_entry amos_functions[] = {...}
```

can be created by using the macro `PHP_FE` for each entry. Now Zend knows which functions this module offers to the user.

Next all necessary information about the module contents must be collected. This information is stored in a structure named `zend_module_entry`. The first entry is a macro and sets the size of the whole zend_module_entry, the number of the Zend module API (`ZEND_MODULE_API_NO`, whether it is a debug build or normal build and if ZTS (Zend Thread Safety) is enabled the fourth value is set to `USING_ZTS`. The second entry contains the name of the module: *amos*. Furthermore the name of the array with all function names (`amos_functions`) must be available to Zend. The fourth and fifth entry are defining functions that are called when initializing (`amos_init`, section 5.1.1) and closing (`amos_free`, section 5.1.2) this module. The next three values are set to `NULL`, as they are not used in this project. The programmer has the possibility to register at this place startup and shutdown functions for page requests and an info function if a footnote should appear at the output of the PHP function `phpinfo()`. The ninth entry identifies the version of this module and the last is a macro setting some remaining internal values.

Another special function for dynamic loadable modules must be considered: the creation of the function `get_module()`. It is called by Zend at load time of the module and has the

```
/* include standard PHP and Amos II headers */
#include "php.h"
#include "callin.h"
#include "php_amos.h"

/* compiled function list, so Zend knows
   what functions are distributed by the Amos II module */
function_entry amos_functions[] = {
  PHP_FE(amos_functionname, NULL)
  ...
  {NULL, NULL, NULL}
}

/* compiled information about Amos II module */
zend_module_entry amos_module_entry = {
  STANDARD_MODULE_HEADER,
  "amos",
  amos_functions,
  ZEND_MINIT(amos_init),
  ZEND_MSHUTDOWN(amos_free),
  NULL, NULL, NULL,
  NO_VERSION_YET,
  STANDARD_MODULE_PROPERTIES
};

/* implement standard "stub" routine to
   introduce Amos II module to Zend */
#ifdef COMPILE_DL_AMOS
  ZEND_GET_MODULE(amos)
#endif

/* Macro for implenting a PHP function */
PHP_FUNCTION(amos_functionname) {
  ...
}
```

Figure 7: Code structure for the Amos II extension

```
PHP_FUNCTION(function_name) {
  /* retrieving the number of PHP arguments */
  int param_count = ZEND_NUM_ARGS();

  /* parsing arguments */
  switch (param_count) {
  /* parse parameters here */
  case x:
    ...
  default:
    WRONG_PARAM_COUNT;
  }

  /* check parameters */

  /* do function action */

  /* return possible values */
}
```

Figure 8: Basic structure for an PHP function

task to pass module information to Zend in order to inform the engine about the module contents.
This function can be implemented by using the macro `ZEND_GET_MODULE`. Since this feature is only required if the module is built as a dynamic extension, the implementation must be surrounded by a conditional compilation statement that can again be seen in figure 7. Thus if this module would be compiled as a built-in extension, the implementation of `get_module` is simply left out.

Now the basic module structure is described but there still remains the layout of an PHP function. The next section considers this point.

3.2.3 PHP function structure

The general structure of an PHP function and some useful macros are described in this section.

First of all the new module function must be initiated with macro `PHP_FUNCTION`. The macro takes the function name as parameter as it can be seen in figure 8. The interior structure of the function starts with initializing all necessary variables for this function. Especially the number of function parameters can be retrieved by the macro `ZEND_NUM_ARGS()`. The next block deals with parsing the parameters with differing between the amount of parameters. Parsing with a fixed parameter count can be done by using a PHP function as follows:

```
int zend_parse_parameters(int num_args TSRMLS_CC, char *type_spec, ...);
```

Where the first argument refers to the number of PHP function arguments. The second defines the parameter types by a string. When parsing a long- and a string-value, the

PHP type	Receive	Send
long	a_getintelem	a_setintelem
double	a_getdoubleelem	a_setdoubleelem
string	a_getstringelem	a_setstringelem
array	a_getseqelem	a_setseqelem

Table 1: Data exchange between PHP and Amos II

string `type_spec` looks like "ls", where 'l' denotes the first parameter to be a long value and 's' represents a string value. Other possibilities can be found in the PHP manual ([8]). Finally there must be a list of initialized variable addresses, where Zend should store the parameter values. If there is an illegal number of arguments in the PHP function, the macro `WRONG_PARAM_COUNT` prints an error message.

Now we can deal with a fixed number of function arguments of a special type. But what to do if a function must deal with one argument having a runtime dependent type? Zend offers a structure type named *zval* for handling this. Every kind of data can be stored in a *zval* variable and additional information about that data, e.g. the type. Type-checking can thus easily be done with the help of constants (e.g. `IS_LONG`) offered by Zend.

The next step consists of checking the parsed parameters. Normally Amos II connections, scans and tuples need to be checked, whether they are still valid. After this the parameters can be edited and/or passed to Amos II functions. For the case of a returning function PHP again offers some macros to return spezial types, e.g. to return a long-value the macro `RETURN_LONG(long value)` should be used.

3.3 Connecting PHP and Amos II types

Till this chapter all explanations concern the structure of the PHP module named *amos*. Now a closer eye must be kept on some internal aspects.

Data exchange between PHP applications and Amos II databases can be done via the *get* and *set* methods of the *callin*-interface from Amos II. Table 1 lists an overview of all basic PHP types and how to receive and send this data to Amos II. The corresponding PHP functions, this project offers to the PHP user, are described in chapter 5.4.

The next section describes how Amos II object identifiers are handled.

3.4 Passing OIDs

The object oriented character of Amos II is already explained in chapter 2.3.3. It is mentioned that a programmer using C must deal with *object handles*. But how to use these object identifiers?

Basically object handles are logical pointers to an Amos II data structure. Any kind of data in Amos II (numbers, strings, arrays, Amos II OIDs and other internal data structures) can be referenced by an object handle. Object handles have the C datatype `oidtype` that is represented by an unsigned integer. They can be declared by using

```
    dcl_oid(o);
```

and after usage they must be freed by calling

```
free_oid(o);
```

And there exists a macro for assigning a new value to a handle:

```
a_setf(<location>, <new value>);
```

where `<location>` is an oidtype that receives a new value `<new value>`. This macro invokes an incremental garbage collector based on reference counting. Thus the old value stored in `<location>` is deallocated if no other location references the old value. It must be mentioned that `free_oid` deallocates the referenced object only if no other location refers to it. Forgetting to free a handle can result in memory leaks as the internal counter might not be reduced.

The question now for this section is how to pass an object to the user. It might simply be done by passing the object handle as an unsigned integer to the PHP script. But this method has some significant disadvantages listed below.

- The PHP user has no idea what is referenced by this integer value. It might be a string, an integer or, what we really want, an OID.
- There is a difference compared to the Amos II command line. There an OID for a surrogate object is printed in a scheme as follows:

  ```
  #[OID <oidnr>]
  ```

  `<oidnr>` is a system maintained object identifier for surrogate objects. When the PHP user wants to print the object, he either prints only the references to the real value, or he would have to call an additional function to transform the object handle into a fitting string. Therefore the object handle must reference a surrogate object, otherwise this transformation would lead into an error.
- An object handle cannot directly be used with a query. The same string, as mentioned in the second item, has to be used. Thus a function call for transforming the object handle to this string is a must.

The other possibility is to convert a surrogate object handle straight into a PHP string with syntax "`#[OID <oidnr>]`" and return it to the PHP script. Integer value `<oidnr>` is the system maintained object identifier for surrogate objects. The value is unique for each object. It must be mentioned that only references to surrogate objects can be transformed, as literal objects have no object identifiers. And thus literal objects such as string and numbers must be converted to the corresponding PHP literal objects. The advantage of this method is that the PHP user need not take care of the internal representation of objects as object handles.

But how to transform a surrogate object handle into this OID string and vice versa?
Two steps are necessary for converting an oidtype to the object string as it can be seen in the code extract of figure 9. Before starting with transformation, an object handle must be initialized and a new value must be assigned to it. First step is to get the OID number from the Amos II system by calling the system C function `a_getid`. A try to convert a literal object would result in an error. After this a string is allocated and the OID number is printed into the object string. Finally this string can be returned to the PHP script.

The other direction, passing an object string from PHP to Amos II, can occur when calling e.g. the function `amos_setobjectelem` (section 5.4.11). Again two steps are necessary to

```
char* make_OID_string(oidtype oid) {
   char *toreturn;
   char buffer[30];
   int id = a_getid(oid, TRUE);
   if (a_errorflag) AMOS_ERROR;
   sprintf(buffer, "#[OID %d]", id);
   toreturn = (char *)emalloc(strlen(buffer)+1);
   strcpy(toreturn,buffer);
   return toreturn;
}
```

Figure 9: Transforming an object handle to an object string

gain an object handle out of an object string. The first thing to do is to retrieve the number, the real OID, out of the given string. For example `1101` is the OID from the object string `#[OID 1101]`. Next the location handle of the OID must be obtained, given its OID number. It can be done by calling the Amos II function

```
a_getobjectno(a_connection c, int oid, int catcherror),
```

which returns the expected location handle.

3.5 Error Management

Another important topic for this project is error management integration of Amos II with the one of PHP. First the Amos II error management is explained, then the PHP error management and finally the integration.

Almost all functions of the external C interface have a flag as parameter that indicates the kind of error handling. This parameter is an integer-value and can accept the values `TRUE` or `FALSE`. As well most of the functions return an error flag that indicates the occurance of an error. An example for such a function is

```
int a_nextrow(a_scan s, int catcherror)
```

with `catcherror` defining the kind of error handling. If this parameter is `FALSE` then Amos II deals with the error and possibly shuts down the system. The other possibilty, `TRUE`, means that the user has to deal with the error on his own, what is done in this project.

What happens if an error occurs? First of all there exists a global variable

```
int a_errorflag;
```

which is `0` if the current Amos II functions ends normally or not equal to zero for the case of an error. Some functions, as the one mentioned above, return this flag, and for other functions this global flag must be investigated. For example like this:

```
if (a_errorflag != 0) {
   /* do some error handling here */
}
```

To receive information about the error, the following three global variables are of use.

- For storing the error number:
  `int a_errno;`
- A string explanation of the error:
  `char *a_errstr;`
- A reference to an Amos II object:
  `oidtype a_errform;`

For the case of an Amos II error, this error must be forwarded to the calling PHP script. Zend offers function

```
zend_error(int type, char *message);
```

for printing a message to the current output (e.g. an HTML page). The first function parameter defines the error type, which can have influence on script execution. There exist different error types in PHP, but for this project only two of them are of interest.

- `E_ERROR` – Signals an error and the script execution terminates immideately.
- `E_WARNING` – Signals a generic warning, but the script execution continues.

Integrating this with the Amos II management system results in two define-statements, one representing a warning and the other an error.

```
#define AMOS_ERROR zend_error(E_ERROR, "Error %d: %s", a_errno, a_errstr);
#define AMOS_WARNING zend_error(E_ERROR, "Error %d: %s", a_errno, a_errstr);
```

3.6 Garbage Collection

Every time a connection is established or queries are sent to Amos II, connection, scan and tuple handles are initialized, and they are occupying memory. After termination of a script or a session this storage has again to be freed. That means, two different ways for dealing with garbage collection have to be considered.

First on script termination a garbage collector function should be called that frees all used memory. Except the memory that is needed for a session, for which the connetion should stay up. The Amos II extension offers a completely automatic method to deal with this kind of garbage collection. The most important thing is to retrieve the connection handles that can be freed. To put in concrete terms, as soon as a new connection is going to be established, a function for closing this connection on script termination has to be registered to Zend. The Amos II extension offers `amos_close` (section 5.2.2) for resetting a connection and for freeing all memory that is correlated to that connection.

The user can register functions that are executed when script process is complete. It can be done by using

```
void register_shutdown_function(callback function);
```

As `register_shutdown_function` is a PHP scripting function, it cannot be called directly from the C code. Here Zend offers a possibility to call and execute user functions by using Zend function `call_user_function_ex`. Figure 10 explains how to to this. First the

```
    zval *fct_shutdown, *fct_aclose, *fctparam;
    zval *retval;
    zval ***params;
    MAKE_STD_ZVAL(fct_shutdown);
    MAKE_STD_ZVAL(fct_aclose);
    MAKE_STD_ZVAL(fctparam);
    ZVAL_STRING(fct_shutdown, "register_shutdown_function", 1);
    ZVAL_STRING(fct_aclose, "amos_close", 1);
    ZVAL_LONG(fctparam, new_con);
    params = (zval ***)emalloc(2*sizeof(zval **));
    params[0] = &fct_aclose;
    params[1] = &fctparam;

    /* register amos_close(new_con); */
    if (call_user_function_ex(CG(function_table), NULL,
        fct_shutdown, &retval, 2, params, 0,
        NULL TSRMLS_CC) != SUCCESS) {
      zend_error(E_WARNING, "Function call failed");
    }
```

Figure 10: Example code for registering a function for garbage collection

function name to be called (`register_shutdown_function`) is stored in a `zval` container `fct_shutdown`. Furthermore this function's parameters (the Amos II extension function to initiate the garbage collection `amos_close` and its parameter, the array position of the used connection) must be stored in a `zval`-array (`params`). Finally a return-value must be allocated again as a `zval` variable and then `amos_close` can be registered.

The second step in garbage collection concerns sessions. A session consists of one or more scripts, while old data is still needed. For example the Amos II extension to PHP should offer the possibility to keep connections alive although script execution has ended. PHP already comprises the handling of sessions which makes it easier to use this behavior in the Amos II extension. The difficult part is deallocating all connections, scans and tuples that are associated with this session.

As it can be seen in figure 5, it is possible to link a connection handle to a session. A session has a unique identifier and it can be obtained by invoking the user function `session_id()`. Similar code to the one for registering shutdown functions is necessary to call this function from C code. Thus each time a new connection is going to be established inside a session, this connection is stored together with the current session identifier.

Now it is up to the user to implement a *session save handler*. A session save handler controls six session functions:

- `open($save_path, $session_name)` – opens a session
- `close()` – closes a session
- `read($id)` – reads the current session data

- `write($id, $sess_data)` – writes the current session data
- `destroy($id)` – destroys a session
- `gc($maxlifetime)` – starts garbage collection after a given time

To make these functions available, set them by using

```
session_set_save_handler(
  "open", "close",
  "read", "write",
  "destroy", "gc"
);
```

An example implementation for a session save handler can be seen in figure 11. It is an extended version to the one offered by [8]. The most important part for collecting garbage are the functions `destroy` and `gc`. There must be a call to the Amos II extension to start clearing memory:

```
amos_gc(session_id());
```

This functions checks each stored connection whether it belongs to the current session. If the session identifiers are the same, deallocation process starts.

Now all connection handles can be freed when script process is complete or a sessions ends. Scan and tuple handles are stored in seperate arrays. As the connection is stored for each scan and tuple handle, a simple loop runs through these arrays and frees all with the connection associated scans and tuples.

4 Performance observations

4.1 Performance

The performance is tested by using

- Amos II Release 6, v7
- Apache HTTP Server 2.0
- PHP 4.3.4

The test machine is a Pentium with 800MHz. In case of a client/server database access the Amos II server runs on the same machine.

During the test a query is sent to the database and the result set is printed. Timestamps are taken before and after query execution and after printing all tuple data of the scan. Thus query execution time, t_{query}, result set printing time, $t_{display}$, and total time, t_{total} can be determined and calculated. Time is measured in the current Unix timestamp with microseconds by calling PHP function *microtime()* (for description see [8]).

A test database, with 10000 person objects having a name, has been created for the test. To test a selection query, one of the persons has an address. Therefore an AmosQL script has been generated which fills the database at Amos II startup with *person* objects having

```
function sess_open($save_path, $session_name) {
  global $sess_save_path, $sess_session_name;
  $sess_save_path = $save_path;
  $sess_session_name = $session_name;
  return(true);
}

function sess_close() {
  return(true);
}

function sess_read($id) {
  global $sess_save_path, $sess_session_name;
  $sess_file = "$sess_save_path\sess_$id";
  if ($fp = @fopen($sess_file, "r")) {
    $sess_data = fread($fp, filesize($sess_file));
    return($sess_data);
  } else {
    return(""); // Must return "" here.
  }
}

function sess_write($id, $sess_data) {
  global $sess_save_path, $sess_session_name;
  $sess_file = "$sess_save_path\sess_$id";
  if ($fp = @fopen($sess_file, "w")) {
    return(fwrite($fp, $sess_data));
  } else {
    return(false);
  }
}

function sess_destroy($id) {
  global $sess_save_path, $sess_session_name;
  amos_gc(session_id());
  $sess_file = "$sess_save_path\sess_$id";
  return(@unlink($sess_file));
}

function sess_gc($maxlifetime) {
  amos_gc(session_id());
  return true;
}

session_set_save_handler(
  "sess_open", "sess_close", "sess_read",
  "sess_write", "sess_destroy", "sess_gc");
session_start();
```

Figure 11: An example implementation for a *session save handler*

Test	Query	DB	t_{query}	$t_{display}$	t_{total}
1	1	*c/s*	1.4849369525909	0.48234820365906	1.96728515625
2	1	*c/s*	1.0670609474182	0.39609503746033	1.4631559848785
3	1	*c/s*	1.5469870567322	0.41786289215088	1.9648499488831
4	1	*c/s*	1.2793309688568	0.38292217254639	1.6622531414032
5	1	*c/s*	1.627240896225	0.3448691368103	1.9721100330353
6	1	*e*	1.3398990631104	0.50183582305908	1.8417348861694
7	1	*e*	0.95206499099731	0.36082601547241	1.3128910064697
8	1	*e*	1.031594991684	0.40175485610962	1.4333498477936
9	1	*e*	1.1383380889893	0.45624804496765	1.5945861339569
10	1	*e*	1.0584740638733	0.49389100074768	1.552365064621
11	2	*c/s*	0.016000032424927	0.00046181678771973	0.016461849212646
12	2	*c/s*	0.015414953231812	0.00045299530029297	0.015867948532104
13	2	*c/s*	0.015239000320435	0.00049495697021484	0.015733957290649
14	2	*c/s*	0.016252994537354	0.00043201446533203	0.016685009002686
15	2	*c/s*	0.015311002731323	0.00045895576477051	0.015769958496094
16	2	*e*	0.011516809463501	0.00043606758117676	0.011952877044678
17	2	*e*	0.011462926864624	0.00046205520629883	0.011924982070923
18	2	*e*	0.0094878673553467	0.00044012069702148	0.0099279880523682
19	2	*e*	0.009490966796875	0.00058412551879883	0.010075092315674
20	2	*e*	0.0096340179443359	0.0004417896270752	0.010075807571411

Table 2: Performance measurement results

a *name* and one of them an *address*. Two queries, a scan and a selection query, have been tested:

1. `select name(p) from person p;`
2. `select livesat(p) from person p where name(p)='Name9975';`

The first query scans the database for all names, which will cause the database and PHP to handle a large result set (10000 names). The display time $t_{display}$ can be expected to be quite high. The second query searches the address of a person with the name *Name9975*. Here the display time can be expected to be small. Both queries are sent via two different connection types, a tight connection – symbolized through an *e* in the *DB* column – and a client/server connection – symbolized through a *c/s*. The resulting times are displayed in table 2.

To summarize table 2 and to draw conclusions the average for each type is calculated:

- Result for Query 1, *c/s* (test cases 1-5)
  $t_{query,avg} = 1.401$
  $t_{display,avg} = 0.404$
  $t_{total,avg} = 1.805$
- Result for Query 1, *e* (test cases 6-10)

$$\begin{aligned} t_{query,avg} &= 1.104 \\ t_{display,avg} &= 0.443 \\ t_{total,avg} &= 1.547 \end{aligned}$$

- Result for Query 2, c/s (test cases 11-15)

$$\begin{aligned} t_{query,avg} &= 0.015 \\ t_{display,avg} &= 0.00046 \\ t_{total,avg} &= 0.015 \end{aligned}$$

- Result for Query 2, e (test cases 16-20)

$$\begin{aligned} t_{query,avg} &= 0.010 \\ t_{display,avg} &= 0.00047 \\ t_{total,avg} &= 0.010 \end{aligned}$$

What is the gain in running Amos II in the PHP address spaces compared to running it client-server? To focus on the first query, there is a speedup of 21% when using an embedded connection instead of a client-server connection. The second query delivers a speedup of 33%. The display time is just as expected. It has a larger influence on the first query, it takes $0.4s$ to print the result set. Definitely no influence on total time has the display time of the second query.

4.2 Debugability

The Amos II extension to PHP is program module that lies between two large projects, PHP and Amos II. This makes the possibility to debug the program quite difficult. It is possible to execute PHP scripts on the command line. With the extern variable

```
extern int trace_interface = TRUE;
```

message logging for Amos II is switched on. Thus it is possible to see messages that are received and sent by an Amos II server.

Another point that shows the difficulty for debugging is that some errors occured only when PHP is used with a web server. The execution of the PHP script resulted in a crash. As Amos II also run as a server, it stayed up. When testing the same script on the command line to debug the script, no error occurred.

4.3 Comparision: PHP versus JSP

JSP (JavaServer Pages) is, similar to PHP, a server side scripting language. First the structure for JSP is explained and after this a comparision between both is done.

4.3.1 JSP structure

JavaServer Pages, short *JSP*, are based on *Java Servlets*. Servlets are Java classes that are used to extend the capability of servers. Through *request-response* architecture a servlet communicates with clients (e.g. browsers). A JSP page is a simple ASCII textfile that contains two kinds of data. First *static template data* that is responsible for creating static content of a webpage. Secondly JSP elements are provided to add a dynamic content to the

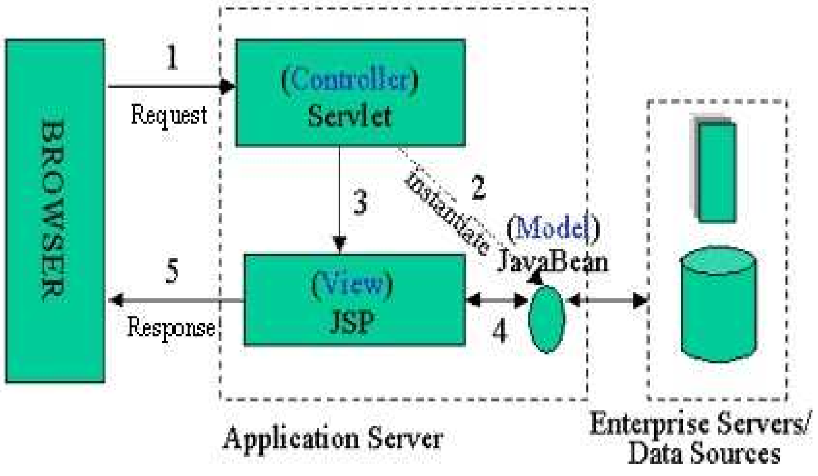

Figure 12: The *JSP Model 2* structure, proposed in [12]

text file. JSP is able to project the full dynamic functionality of Java Servlets Technology, but provides a more natural approach to create static content [11].

The combination of all described technologies results in the *JSP Model 2* structure, illustrated by figure 12. Let's follow a request initiated by a browser. The request is received by the Servlet that works as a controller. The data transfered through the request has to be stored. Therefore a JavaBean is instantiated that contains the sent data. Additionally, data from exernal data sources may be loaded into this JavaBean instance. As a third step the controller starts the view generation, in other words, an HTML page is generated out of the JSP page. JSP is used for the so called *presentation layer*. For binding the dynamic data into the generated HTML page, data from the JavaBean instance is used. As soon as an HTML has been generated, it is returned as a response to the request to the browser, which displays the dynamic generated web page.

4.3.2 General comparison

PHP and JSP are not entirely different, but they exist for a different purpose. While PHP does everything concerning processing, logic and layout in the same page, JSP should only be used on the presentation layer. The following list gives a comparison translated from [13].

1. Common features of PHP and JSP
    - server side scripting languages for HTML integreation
    - requires an HTTP servermodule with JSP/PHP runtime environment
    - pages must be read and interpreted
    - offer session handling
2. JSP
    + independent of operating system
    + independent of server

+ programming is made in Java (no new programming language)
+ access to the complete Java-API and Java-classes
+ supports separation of layout and program logic
+ unique access on almost all relational database systems via JDBC
– large use of resources for Java runtime environment

3. PHP

+ typeless variables and dynamic array structures simplify web programming
+ automatic processing of formulars
+ high error tolerance, especially for programming with variables
+ contains a great amount of modules (e.g. for data compression, pay services, XML-processing)
+ offers a lot of macros and functions to write external modules
– supported DB-functionality is dependent on the database; specific functions for each database system are existing
– a large part of program logic has to be implemented as an external module in C/C++

4.3.3 Amos II specific comparison

In this chapter an Amos II specific comparison between PHP and JSP is done. As JSP is built on Java, the PHP extension is compared with the external Java interface [6].

1. Garbage Collection
   - The Amos II extension to PHP offers a partly automatic garbage collection to the user. For scripts, executed like CGI scripts, the user need not care about garbage collection. Using Amos II funtions in combination with sessions a *session save handler* has to implemented by the user. See chapter 3.6 for more details.
   - Garbage collection in Java is secured through the already built in garbage collector.
2. Error Management
   - All Amos II errors are forwarded to the Error Management of PHP. These errors or warnings are then printed to the current output stream.
   - There exists an *AmosException* in Java that can be caught by using a *try-catch* clause.
3. Connections, scans and tuples are treated in dependence of the programming language almost the same way.
4. As Java supports overloading on method arguments, there exists only one function, *setElem*, to set values to tuples. There exists no overloading for return values in Java, therefore different get-functions exist for each type. In PHP it is possible to have only one function for setting and one for getting values. As explained in chapter 7, the

interface can be further simplified by using PHP arrays. In this way all set and get methods can be eliminated which makes the PHP interface significantly simpler than the Java interface.

5. Function calls

   - Every time an Amos II function is called from Java, the function name has to be passed as argument.
   - In C the function to call is specified by an OID representing the Amos II function to be called. Right now, parameters have to be passed as a tuple handle. Based on experiences from the basic interface an improved one is proposed using PHP arrays instead of tuples. The proposed improvements are explained in chapter 7.

   The same characteristic is valid for adding, setting and removing stored function data.

5 Function reference

This chapter gives a complete overview of all implemented functions to access Amos II databases from PHP scripts. For full functionality, some entries in file *php.ini* must be added.

1. Make PHP know the Amos II extension:

   ```
   extension=php_amos.dll
   ```

   This entry makes the Amos II extension available to PHP. Be sure not to use the semicolon in front of the line! Thus the extension will automatically be loaded at PHP startup.

2. Define some module initialization parameters:

   ```
   [amos]
   ; file for Amos II initialization
   amos.init_file = $AMOS_ROOT\bin\amos2.dmp

   ; array sizes at module initialization
   amos.max_cons = 10
   amos.max_scans = 20
   amos.max_tuples = 30
   ```

   The first entry is an image file that is used to initialize Amos II for the first time. Normally file *amos2.dmp* should be declared. The other three entries are defining the arraysize for storing connection, scan and tuple handles at startup. If the average number of accesses (database connections) is known, these values can be adjusted. It will have no influence on the programs behaviour and little on the performance as the arrays will automatically increased when running out of memory. Notice that there should be more scans than connections as every connection can hold several scans. The same is valid for tuples.

This chapter is organized as follows. First functions concerning the module are explained. Next primitive functions starting with connecting to Amos II, scan handling and ending with the tuple interface. Leaving the basic functions, the *fast-path* interface, object creation and finally transaction control is explained.

Before studying the functions, some conventions as follows should be considered.

1. A *connection* is an index to the array of Amos II connection handles.
2. A *scan* is an index to the array of Amos II scan handles.
3. A *tuple* is an index to the array of Amos II tuple handles.

A PHP programmer uses all indices via PHP variables of type long.

So each time a connection, scan or tuple is passed to or returned by a function, it is only the index that is passed or returned. It has no influence on the meaning of the described function, but this reference is much easier to read.

5.1 Module functions

As soon as PHP starts up, the Amos II extension is automatically loaded. With it *amos_init* is called. On module shutdown *amos_free* is called. Notice here that these two functions cannot be accessed by PHP users. Furthermore it is possible to call a garbage collection routine, *amos_gc*, out of a PHP script.

5.1.1 `amos_init`

Three things are done in this function. First the Amos II system is started using

```
a_initialize(char *image, int catcherror);
```

with *image* being the file for initializing Amos II. In addition a default database that store for example information about existing types and functions is loaded. The *image*-value can be defined in *php.ini* by setting *amos.init_file*. Zend function

```
ini_get('amos.init_file');
```

retrieves this value.

The second step consists of allocating memory to store connection, scan and tuple handles. Therefore three arrays of structure *amos_con*, *amos_scan* and *amos_tuple* are allocated. For every entry the *status* is set to `0`, with the meaning that no handle is stored at this position. Furthermore *con_index* is set to `-1` for all scans and tuples. As soon as a scan or a tuple is stored, the index to the connection handle under that the scan or tuple has been created is assigned to *con_index*. If one of the arrays runs out of storage space, a *realloc*-operation resizes the array.

Finally some constants are registered to Zend:

- `AMOS_INTEGERTYPE` – marks an object of type *integer*.
- `AMOS_REALTYPE` – marks an object of type *double*.
- `AMOS_STRINGTYPE` – marks an object of type *string*.

- `AMOS_ARRAYTYPE` – marks an object of type *array*.
- `AMOS_OIDTYPE` – marks an object of type *object*.

The scope of these constants is dependent on the module's lifetime.

5.1.2 `amos_free`

During module shutdown all three arrays that are storing connection, scan and tuple handles are freed.

5.1.3 `amos_gc`

Frees memory used by a connection or a session.

Syntax:

```
void amos_gc(long amos_con);
void amos_gc(string session_id);
```

Description:

Not only on module shutdown, but also at the end of scripts or sessions, garbage must be collected and used memory freed. It is possible for the user to invoke this function whenever he wants, but it must be invoked at the points chapter 3.6 lists.

This method can receive two different parameter types.

- The first possibility is calling *amos_gc* with a connection *amos_con*. All tuples and scans are searched and deallocated that are belonging to that connection. The connection is closed and the handle is deallocated.
- A string, representing a session identifier (*session_id*), is the second possibility. There is a loop that searches all connections that are correlated to the current session. If a connection is found, it will be handled the same way as described above.

5.2 Connection interface

This interface offers PHP functions for connecting and disconnecting to Amos II databases.

5.2.1 `amos_connect`

A connection to Amos II is established and an index to a connection handle is returned to the user.

Syntax:

```
long amos_connect(string dbname);
```

Description:

Before accessing an Amos II database the user must open a connection by using function *amos_connect*. The database can be specified by argument *dbname*. In dependance of the parameter value two different kinds of connections can be established:

1. The first possibility is a *tight connection* to an Amos II database. It can be established by passing an empty string.

   ```
   $mytightcon = amos_connect("");
   ```

   Now Amos II runs as an embedded system inside the PHP application.

2. The second possibility is to open a connection to an existing Amos II database by passing a non-empty string to *amos_connect*:

   ```
   $mycon = amos_connect("mydatabasename");
   ```

   This will work like a *client-server connection*, with Amos II running as server (Amos II peer). Then a free amount of scripts (clients) can access the database.

As soon as a connection handle is created it will be stored in an internal global array. The user can access this connection by a long-value that points to the array position holding the connection handle.

Now garbage collection has to be prepared. Therefore function *amos_gc* is registered to Zend with the connection as parameter. At the end of the current PHP script it will be executed and thus automatically free all to the connection correlated tuples and scans and finally the connection itself. In case of a session the session identifier is stored in combination with the connection handle. If a correct working session save handler is implemented for the script – which has to be done by the user – everything, corresponding to the connection, will be deallocated as soon as the session ends. See chapter 3.6 for more details!

If no error occured, the index that holds the new handle is returned to the user, otherwise '`-1`' is returned.

5.2.2 `amos_close`

Closes a connection and initiates garbage collection.

Syntax:

```
void amos_close(long amos_con);
```

Description:

Function *amos_close* is offered to the user to control garbage collection in addition to the other methods that are described so far. If the passed index, *amos_con* contains a valid connection handle, the garbage collection routine is started. All tuples and scans belonging to that connection are deallocated. At last the connection handle is freed. The array index is signaled to be free by setting a flag.

5.3 Scan interface

The scan interface offers PHP functions to send queries to Amos II peers and to run through returned result sets. Figure 13 show how the scan interface can be used.

```
/* sending query */
$current_scan = amos_query($con, $query);

/* handling result set */
while (!amos_eos($current_scan)) {
  amos_getrow($current_scan, $tpl);
  ...// code to retrieve elements of tuple 'tpl'

  amos_next($current_scan);
}
```

Figure 13: Example code for using the scan-interface.

5.3.1 amos_query

Sends queries to Amos II databases and returns scans.

Syntax:

```
long amos_query(long amos_con, string query);
```

Description:

Any kind of queries written in AmosQL can be passed to this function via parameter *query*. In dependance of connection *amos_con* the query is forwarded to an Amos II database. The Amos II system generates a result set and fills it with tuples. Then this query result is delivered to the Amos II module and is stored in a global array. In combination with the current scan the index to connection *amos_con* is stored. Thus the garbage collection routine is able to determine all scans that belong to a connection.

If no error occurs, the index that holds the new handle is returned to the user, otherwise '`-1`' is returned.

The following example returns all function names currently stored in an Amos II system:

```
$query = "select name(f) from function f;";
$scan_index = amos_query($current_con, $query);
```

Notice that all AmosQL statements must end with a semicolon!

5.3.2 amos_closescan

Closes a scan.

Syntax:

```
void amos_closescan(long amos_scan);
```

Description:

Function *amos_closescan* frees the scan handle stored at index *amos_scan*. The array index is signaled to be free by setting a flag.

5.3.3 `amos_eos`

Checks for more rows (tuples) in a scan.

Syntax:

```
bool amos_eos(long amos_scan);
```

Description:

It is checked if the current curser of scan *amos_scan* points to a valid tuple. If so, `TRUE` is returned, otherwise `FALSE`. This function is well used inside a while-loop as shown in figure 13. In that way all tuples of a scan are taken into account.

5.3.4 `amos_getrow`

Copies the current row into a tuple handle.

Syntax:

```
long amos_getrow(long amos_scan, long amos_tuple[, array empty_array]);
```

Description:

Before elements of data can be retrieved, it must be copied from the current curser position of scan *amos_scan* into tuple *amos_tuple*. The tuple can be created by using PHP function `amos_createtuple` (section 5.4.2).

The index of the tuple-handle with new data is returned. But it is the same index like the parameter *amos_tuple*.

An example code is shown in figure 13.

As an additional feature, *amos_getrow* can be called by passing an empty array. This function initializes this array and fills it recursively with the data of the current row. Why recursively? An Amos II tuple can contain arrays, thus the array is stored inside the main array. And this has to be done recursively. The PHP code gets very simple by using this method. No functions calls of the tuple-interface have to be done. The array is filled with all the data of the row, and the user can run through the array to read the data. This function still returns the used tuple, that contains the same data and can be accessed by functions from the tuple-interface. The following PHP code gives an example:

```
$row = array();
$tuple = amos_getrow($scan, $tuple, $row);
/* PHP method for printing arrays */
print_r($row);
```

5.3.5 `amos_next`

Advances the scan curser position one step forward.

Syntax:

```
void amos_next(long amos_scan);
```

Description:

```
$i = 0;
$scan = amos_query($con,
   "select name(f) from function f;");
$tpl = amos_createtuple($con);
while (!amos_eos($scan)) {
  amos_getrow($scan, $tpl);
  $val = amos_getstring($tpl, 0);
  print "row $i:  $val";
  amos_next($scan);
  $i = $i + 1;
}
```

Figure 14: Example PHP code for using the tuple- and scan-interface

The current tuple is set to the next tuple in scan *amos_scan.* Notice that the scan handle will automatically be closed by Amos II when all tuples have been read or another query result is assigned to the scan.

An example code is shown in figure 13.

5.4 Tuple interface

The tuple-interface offers PHP functions to set and get data to and from a tuple. Figure 14 shows an example how to use the tuple interface in combination with the scan interface.

5.4.1 `amos_getarity`

Returns the arity of a tuple.

Syntax:

```
long amos_getarity(long amos_tuple);
```

Description:

Tuples are representing a complete row of a query result. This functions returns the number of elements in tuple *amos_tuple.* All elements in a tuple are enumerated beginning with index '0' for the first element.

5.4.2 `amos_createtuple`

Creates an empty tuple.

Syntax:

```
long amos_createtuple(long amos_con[, long arity]);
```

Description:

Before using a tuple, it must be created by this function. Either a tuple with an undefined length or a tuple with a given *arity* can be created. If the tuple is used with function *amos_getrow*, it is not necessary to pass an arity to create a tuple. It will automatically be filled with all necessary elements. For garbage collection tuples must belong to a connection defined by parameter *amos_con*. This connection is stored in combination with the tuple handle.

If no error occurs, the index that holds the new handle is returned to the user, otherwise '`-1`' is returned.

The following example creates a new tuple with the arity of another one:

```
$new_tpl = amos_createtuple($con, amos_getarity($old_tpl));
```

5.4.3 `amos_closetuple`

Closes a tuple.

Syntax:

```
void amos_closetuple(long amos_tuple);
```

Description:

Function *amos_closetuple* frees the tuple handle stored at index *amos_tuple*. The array index is signaled to be free by setting a flag.

5.4.4 `amos_getint`

Returns an integer value.

Syntax:

```
long amos_getint(long amos_tuple, long pos);
```

Description:

Use function *amos_getint* to gain an integer value. It takes the element value of tuple *amos_tuple* at position *pos* and returns the integer value. The specified element must hold an integer, otherwise an error message is printed and script execution is stopped.

The code line as follows gives an example on how to gain an integer value at position '`1`' from tuple '`tpl`':

```
$result_int = amos_getint($tpl, 1);
```

5.4.5 `amos_setint`

Assigns an integer value to an element of a tuple.

Syntax:

```
void amos_setint(long amos_tuple, long pos, long value);
```

Description:

Function *amos_setint* stores a *value* of type long to the element in tuple *amos_tuple* at position *pos*. This function is not intended to set an integer value to a result tuple, e.g. result of *amos_getrow*. The reaction would be a fatal error. But it is possible to set the element value of a tuple more than once, even with different types.

5.4.6 `amos_getdouble`

Returns a double precision floating point number.

Syntax:

```
double amos_getdouble(long amos_tuple, long pos);
```

Description:

Use function *amos_getdouble* to fetch a double value. It takes the element value of tuple *amos_tuple* at position *pos* and returns the double value. The specified element must hold a double, otherwise an error message is printed and script execution is stopped.

The code line as follows gives an example on how to gain a double value at position '`1`' from tuple '`tpl`':

```
$result_double = amos_getdouble($tpl, 1);
```

5.4.7 `amos_setdouble`

Assigns a double value to an element of a tuple.

Syntax:

```
void amos_setdouble(long amos_tuple, long position, double value);
```

Description:

Function *amos_setdouble* stores a *value* of type double to the element in tuple *amos_tuple* at position *pos*. This function is not intended to set a double value to a result tuple, e.g. result of *amos_getrow*. The reaction would be a fatal error. But it is possible to set the element value of a tuple more than once, even with different types.

5.4.8 `amos_getstring`

Returns a string.

Syntax:

```
string amos_getstring(long amos_tuple, long pos);
```

Description:

Use function *amos_getstring* to gain a string value. It takes the element value of tuple *amos_tuple* at position *pos* and returns the string value. The specified element must hold a string, otherwise an error message is printed and script execution is stopped.

The code line as follows gives an example on how to gain an string value at position '`1`' from tuple '`tpl`':

```
$result_str = amos_getstring($tpl, 1);
```

5.4.9 `amos_setstring`

Assigns a string value to an element of a tuple.

Syntax:

```
void amos_setstring(long amos_tuple, long position, string value);
```

Description:

Function *amos_setstring* stores a *value* of type string to the element in tuple *amos_tuple* at position *pos*. This function is not intended to set a string value to a result tuple, e.g. result of *amos_getrow*. The reaction would be a fatal error. But it is possible to set the element value of a tuple more than once, even with different types.

5.4.10 `amos_getobjectelem`

Returns the OID of an Amos II surrogate object as a string.

Syntax:

```
string amos_getobjectelem(long amos_tuple, long position);
```

Use function *amos_getobjectelem* to gain an OID string. It takes the element value of tuple *amos_tuple* at position *pos* and returns the object identifier string. The returned OID string has a structure as follows:

```
#[OID 1101]
```

with `1101` being the actual identifier number. This function only retrieves the OID of surrogate objects. Literals have no OID. Use the other *get*-methods to get literal objects, e.g. to get a string use *amos_getstring*. The code line as follows gives an example on how to gain an OID string at position '`1`' from tuple '`tpl`':

```
$result_oid = amos_getobjectelem($tpl, 1);
```

The result string can simply be used in queries written in AmosQL:

```
select name(p) from person p
where address(p)=$address_oid;
```

This query delivers the name of the person that lives at *$address_oid*.

5.4.11 `amos_setobjectelem`

Assigns an OID to an element of a tuple.

Syntax:

```
void amos_setobjectelem(long amos_tuple, long pos, string oid);
```

Description:

Function *amos_setoblectelem* stores an object with the corresponding OID string *oid* to an element in tuple *amos_tuple* at position *pos*. This function is not intended to set an OID string to a result tuple, e.g. result of *amos_getrow*. The reaction would be a fatal error. But it is possible to set the element value of a tuple more than once, even with different types.

5.4.12 `amos_getseqelem`

Returns an index to a tuple that represents the sequence.

Syntax:

```
long amos_getseqelem(long amos_tuple, long pos);
```

When tuple *amos_tuple* holds a sequence element at position *pos*, the index to a new tuple, which represents the sequence, is returned by function *amos_getseqelem*.

To ensure garbage collection, the connection index stored with tuple *amos_tuple* is used to create the new tuple. Thus in case of a disconnect, enough information exists to free this tuple.

5.4.13 `amos_setseqelem`

Assigns a sequence to a tuple.

Syntax:

```
void amos_setseqelem(long amos_tuple, long position, long sequence_tuple);
```

Description:

Function *amos_setseqelem* stores a sequence, *sequence_tuple*, to the element in tuple *amos_tuple* at position *pos*. This function is not intended to set a sequence to a result tuple, e.g. result of *amos_getrow*. The reaction would be a fatal error. But it is possible to set the element value of a tuple more than once, even with different types.

5.4.14 `amos_elemsize`

Returns the size of an element in a tuple.

Syntax:

```
long amos_elemsize(long amos_tuple, long pos);
```

Description:

In dependence of the element type at position *pos* of tuple *amos_tuple*, the size of that element is returned. In case of a string, the string size is returned. If the element holds a sequence, the number of elements in that sequence is returned.

5.5 Fast-path function calling

All functions described so far deal with the embedded query call. Queries can be sent to Amos II databases and a result set is returned. This result can be explored by the tuple-interface. The *fast-path* interface permits to call Amos II functions from PHP without sending a query. The result of a fast-path function call is again a scan handle. It can be treated the same way like the result sets from the embedded query call.

During this chapter PHP functions are introduced to call functions and to add, set and

```
    /* create a new tuple for storing function arguments */
    $argl = amos_createtuple($con, 1);
    amos_setobjectelem(argl, 0, $person_oid);

    /* get function id */
    $fct_id = amos_getfunction($con, "person.name->charstring");

    /* execute function */
    $scan = amos_callfunction($con, $argl, $fct_id);

    /* handle scan result by using tuple interface */
    ...
```

Figure 15: Example PHP code for using the fast-path interface

remove data of a function. But how to pass parameters to the called functions? It can be done by using *argument lists* that contains all arguments. A tuple represents this list and must be allocated with the correct arity (number of parameters) by using

```
$tpl = amos_createtuple($con, arity);
```

Now this tuple can be filled with argument values by using tuple update functions. The first parameter should be stored at position '`0`' (the first tuple element), etc. Finally this list can be passed as parameter to the corresponding PHP function `amos_callfunction`.

Figure 15 shows an example on how to retrieve the name of a person by using the fast-path interface.

Concerning the speed, this interface is much faster than the embedded query call as no query parsing and optimization has to be done.

5.5.1 `amos_getfunction`

Retrieves the OID string of a function.

Syntax:

```
string amos_getfunction(long amos_con, string function_name);
```

Description:

Before calling *amos_getfunction* the OID of the called function is needed. Normally the user has only knowledge of the function name (e.g. '`FUNCTION.ARGUMENTS->VECTOR`'). Thus a function is necessary to retrieve the OID from the function name.

Function *amos_getfunction* takes a connection, *amos_con*, and an Amos II function name, *function_name*, as parameters. The return value is an OID string that represents the function name.

5.5.2 `amos_gettype`

Returns the OID string of a type.

Syntax:

```
string amos_gettype(long amos_con, string typename);
```

Description:

Function *amos_gettype* is the analogous to *amos_getfunction* for types. It computes in dependence of the connection *amos_con* the OID string of a specified type, *typename*. The OID string of a type is for example necessary for creating and deleting new objects (see *amos_createobject* and *amos_deleteobject* at chapter 5.6).

5.5.3 `amos_callfunction`

Calls a specified function.

Syntax:

```
long amos_callfunction(long amos_con, string function_oid, long fct_argl);
```

Description:

With Amos II connection *amos_con*, an Amos II function, specified by OID string *function_oid*, is called by using this PHP function. Parameters of the Amos II function must be stored in a tuple and passed through parameter *fct_argl*. The return value is an index pointig to the result scan.

Figure 15 shows an example that calls an Amos II function to retrieve names of all persons.

5.5.4 `amos_addfunction`

Adds new data to a function.

Syntax:

```
void amos_addfunction(long amos_con, string function_oid,
   long argl, long resl);
```

Description:

Function *amos_addfunction* adds new data to an Amos II function. In dependence of connection *amos_con* new data is added to function *function_oid*. The data of a function is definite specified by its arguments and by its result set. All arguments are stored in an argument list (a tuple). The result values are again stored in a tuple. For both, an index to the prevailing tuple (*argl* and *resl*) is passed to this function.

The following code gives an example that sets a name, *Marc*, to an empty person object *person_oid*.

```
/* add new data to function 'person.name->charstring' */
amos_setobjectelem($argl, 0, $person_oid);
$name = "Marc";
amos_setstring($resl, 0, $name);
amos_addfunction($con, $fct_oid, $argl, $resl);
```

5.5.5 `amos_setfunction`

Sets the value of a function.

Syntax:

```
void amos_setfunction(long amos_con, string function_oid,
    long argl, long resl);
```

Description:

Assigns a new value to an Amos II function. Concerning parameters an index to a connection handle *amos_con*, the function handle *function_oid*, a tuple *argl* containing the argument list and a tuple *resl* that stores the result value are needed. PHP function `amos_setfunction` assigns *resl* as the result of applying *function_oid* on the argument tuple *argl*. The difference to function *amos_addfunction* is the replacement of an old existing value, while the addfunction adds completely new data.

5.5.6 `amos_remfunction`

Removes data from a function.

Syntax:

```
void amos_remfunction(long amos_con, string function_oid,
    long argl, long resl);
```

Description:

Removes data from an Amos II function. In dependence of connection *amos_con* data is removed from function *function_oid*. The data to remove is described by the argument list, represented by tuple *argl*, and the result tuple *resl*.

5.6 Object creation and deletion

The Amos II extension offers methods for creating and deleting objects.

5.6.1 `amos_createobject`

Creates a new object.

Syntax:

```
string amos_createobject(long amos_con, string type);
```

Description:

Function *amos_createobject* creates a new object of a specified type. The index to a connection handle, *amos_con*, and the type as a string, *type*, are taken as parameters. With this information a new surrogate object is created and the OID string of that created object is returned to the user.

For creating a new object of type `person` call:

```
$newobj = createobject($con, "person");
```

Variable *$newobj* holds the returned OID string that identifies the new object.

5.6.2 `amos_deleteobject`

Deletes a surrogate object.

Syntax:

```
void amos_deleteobject(long amos_con, string type_oid);
```

Description:

For deleting an existing object the index to the connection handle *amos_con* and the OID string that identifies this object is necessary.

5.7 Transaction control

These transaction control primitives have only effect on tight connections. Using an embedded database the user can send a lot of operations to the Amos II system. Only when the *commit*-function is called, changes will be fixed. In case of a *rollback*, all changes disappear.

There exists an *autocommit* for client-server connections. After the execution of every operation data changes are fixed and stored in the database. It is thus useless to call transaction control functions explicitly.

5.7.1 `amos_commit`

Commits a transaction.

Syntax:

```
void amos_commit(long amos_con);
```

Description:

All operations for the specified connection, *amos_con*, are fixed with calling this function. It will have effect only on embedded databases.

5.7.2 `amos_rollback`

Aborts a transaction.

Syntax:

```
void amos_rollback(long amos_con);
```

Description:

All operations to connection *amos_con* that are executed after the last *commit* are cancelled with this operation. It will have effect only on embedded databases.

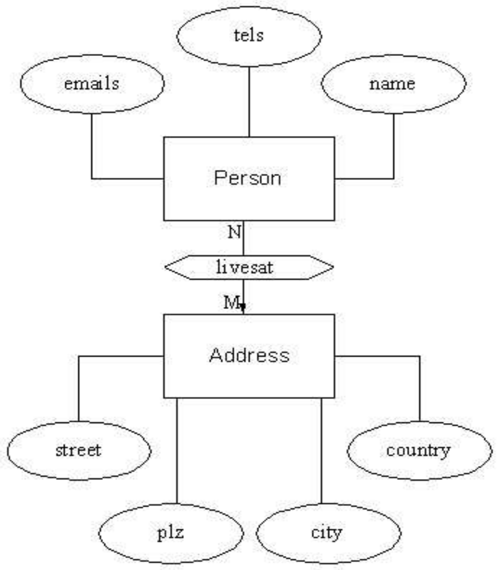

Figure 16: Entity-Relationship diagram for the Addressbook database

6 Example Program: The Addressbook

This chapter describes an example database in Amos II and explains PHP sites using this database.

6.1 The database

This chapter describes the database for the addressbook. The ER-diagram is shown in figure 16. Figure 17 shows the corresponding AmosQL script that defines the database.

6.2 The program

The program should have several properties.

- It should be able to display all persons with all information correlated to each person.
- It should be possible to add a new person to the database.
- It should be possible to edit data of an existing person.
- It should be possible to delete an existing person.
- The connection should stay up for a complete session.

These features are realized in three PHP scripts. First of all there is a file *index.php* which displays all information of all persons in a table. Script *new.php* offers text fields for entering data for a new person. And finally file *edit.php* offers filled text fields for editing data for a person. The connection is stored in the global PHP session array with the keyword *con* (`$_SESSION['con']`). Thus the connection stays up and is accessible in other PHP scripts.

```
    create type person;
    create function name(person)->charstring as stored;
    create function emails(person)->vector of charstring as stored;
    create function tels(person)->vector of charstring as stored;

    create type address;
    create function street(address)->charstring as stored;
    create function plz(address)->integer as stored;
    create function city(address)->charstring as stored;
    create function country(address)->charstring as stored;

    create function livesat(person)->address as stored;
```

Figure 17: Database definition for the addressbook

To generate the overview of all persons the query as follows is used.

```
select p, name(p) from person p;
```

With this query the OID string and the name of that person is retrieved out of the database. If there is a person object without a name, it won't be displayed. It is still left to retrieve the email addresses, the telefone numbers and the addresses from that person. To test different methods, all email addresses of one person are fetched by using the fast-path interface:

```
$fct_emails = amos_getfunction($_SESSION['con'],
   "PERSON.EMAILS->VECTOR-CHARSTRING");
$emails_scan = amos_callfunction($_SESSION['con'], $fct_emails,
   $person_tpl);
$emails_tpl = amos_createtuple($_SESSION['con']);
amos_getrow($emails_scan, $emails_tpl);
$emails = amos_getseqelem($emails_tpl, 0);
for ($i=0; $i<amos_getarity($emails); $i++) {
  $email = amos_getstring($emails, $i);
  print "$email";
}
amos_closetuple($emails);
amos_closetuple($emails_tpl);
amos_closescan($emails_scan);
```

The telefone numbers are fetched by sending a query to Amos II:

```
$tels_scan = amos_query($_SESSION['con'],
   "select tels(p) from person p where p=$pid;");
$tels_tpl = amos_createtuple($_SESSION['con']);
amos_getrow($tels_scan, $tels_tpl);
$tels = amos_getseqelem($tels_tpl, 0);
for ($i=0; $i<amos_getarity($tels); $i++) {
  $tel = amos_getstring($tels, $i);
  print "$tel";
}
```

```
amos_closetuple($tels);
amos_closetuple($tels_tpl);
amos_closescan($tels_scan);
```

The addresses are fetched by using both methods, the fast-path interface to retrieve the address objects and the query interface to retrieve the street, the postal code and the city name of the specified address object. The following code fragment is responsible for these actions:

```
/* retrieve address objects */
$fct_livesat = amos_getfunction($_SESSION['con'], "PERSON.LIVESAT->ADDRESS");
$ad_scan = amos_callfunction($_SESSION['con'],
   $fct_livesat, $person_tpl);
$ad_tpl = amos_createtuple($_SESSION['con']);
$ad_oid = amos_getobjectelem(amos_getrow($ad_scan, $ad_tpl), 0);
amos_closetuple($ad_tpl);
amos_closescan($ad_scan);
/* retrieve street, postal code and city */
$ad_scan = amos_query($_SESSION['con'],
   "select street(a), plz(a), city(a) from address a where a=$ad_oid;");
$ad_tpl = amos_createtuple($_SESSION['con']);
amos_getrow($ad_scan, $ad_tpl);
$street = amos_getstring($ad_tpl, 0);
$plz = amos_getint($ad_tpl, 1);
$city = amos_getstring($ad_tpl, 2);
print "$street, $plz $city";
amos_closetuple($ad_tpl);
amos_closescan($ad_scan);
```

For adding a new person the user is able to switch to *new.php*, where text fields as follows can be filled. E-Mails and telefone numbers will be stored in an Amos II vector. For sending the data back to *index.php*, the post method from PHP is used.

- Name
- E-Mail
- E-Mail (2)
- Telefone
- Telefone (2)
- Street
- Postal code
- City

Additionally a PHP session variable, *action*, is created that indicates the action at the beginning of script *index.php*. For adding a new person *action* is set to '*new*'. Back to the overview, the posted data is read and added to the database by using the fast-path interface.

First a new person object has to be created. Then PHP function *amos_addfunction* (5.5.4) is used to add the new data. A code extract is given that will add all email addresses:

```
/* create a new person object */
$p_type = amos_gettype($_SESSION['con'], "person");
$new_person = amos_createobject($_SESSION['con'], $p_type);
$new_person_tpl = amos_createtuple($_SESSION['con'], 1);
amos_setobjectelem($new_person_tpl, 0, $new_person);
/* check email entries */
if (isset($_POST['email'])) {
   $data;
   if (isset($_POST['email2'])) {
      /* two given email entries => tuple arity = 2 */
      $data = amos_createtuple($_SESSION['con'], 2);
      amos_setstring($data, 0, $_POST['email']);
      amos_setstring($data, 1, $_POST['email2']);
      unset($_POST['email']);
      unset($_POST['email2']);
   }
   else {
      /* one given email address => tuple arity = 1 */
      $data = amos_createtuple($_SESSION['con'], 1);
      amos_setstring($data, 0, $_POST['email']);
      unset($_POST['email']);
   }
   /* add function data */
   $data_seq = amos_createtuple($_SESSION['con'], 1);
   amos_setseqelem($data_seq, 0, $data);
   amos_addfunction($_SESSION['con'], $fct_emails, $new_person_tpl, $data_seq);
   amos_closetuple($data);
   amos_closetuple($data_seq);
}
else if (isset($_POST['email2'])) {
   /* one given email address => tuple arity = 1 */
   $data = amos_createtuple($_SESSION['con'], 1);
   amos_setstring($data, 0, $_POST['email2']);
   /* add function data */
   $data_seq = amos_createtuple($_SESSION['con'], 1);
   amos_setseqelem($data_seq, 0, $data);
   amos_addfunction($_SESSION['con'], $fct_emails, $new_person_tpl, $data_seq);
   amos_closetuple($data);
   amos_closetuple($data_seq);
   unset($_POST['email2']);
}
```

For changing the data of an existing person a similar page, *edit.php*, is displayed. To specify the person to change the OID string must be passed to this page. There all text fields contain the old values that can be edited. The *action* variable is set to '*edit*'. Via the post method the edited data is sent back to the overview script, where it is read and the changes are fixed in the database. Therefore function *amos_setfunction* (5.5.5) from the fast-path interface is used. An example code that sets a new value for a telefone number is shown below. It is assumed that there already exists some telefone numbers for this person, so no new function entry has to be created. Otherwise a function call would have to be done to check for an existing entry. Variable *person_tpl* is a tuple that contains the existing person object which data is going to be changed.

```
/* check for telefone entries */
if (isset($_POST['telefone'])) {
   if (isset($_POST['telefone2'])) {
      /* two given telefone numbers => tuple arity = 2 */
      $data = amos_createtuple($_SESSION['con'], 2);
      amos_setstring($data, 0, $_POST['telefone']);
      amos_setstring($data, 1, $_POST['telefone2']);
      unset($_POST['telefone']); // unset handled data
      unset($_POST['telefone2']);
   }
   else {
      /* one given telefone number => tuple arity = 1 */
      $data = amos_createtuple($_SESSION['con'], 1);
      amos_setstring($data, 0, $_POST['telefone']);
      unset($_POST['telefone']);
   }
   /* set function data */
   amos_setfunction($_SESSION['con'], $fct_tels, $person_tpl, $data);
   amos_closetuple($data);
}
else if (isset($_POST['telefone2'])) {
   /* one given telefone number => tuple arity = 1 */
   $data = amos_createtuple($_SESSION['con'], 1);
   amos_setstring($data, 0, $_POST['telefone2']);
   /* set function data */
   amos_setfunction($_SESSION['con'], $fct_tels, $person_tpl, $data);
   amos_closetuple($data);
   unset($_POST['telefone2']);
}
else {
   /* no given telefone number => remove old values ($old_data) */
   amos_remfunction($_SESSION['con'], $fct_tels, $person_tpl, $old_data)
}
```

The last possible user action is removing an existing person. No new script is necessary - file *index.php* is reloaded. The person to delete is specified by an OID string. The remove action is defined by an unset *action* variable and a set post variable `$_POST['person']`. With function *amos_remfunction* (5.5.6) from the fast-path interface all data can be removed. Below the code extract is displayed that deletes a given person object. It is not necessary to delete all function entries on its own, it is enough to delete only the person object.

```
if (!isset($_SESSION['action']) && isset($_POST['person'])) {
   amos_deleteobject($_SESSION['con'], $_POST['person']);
```

```
    unset($_POST['person']);
}
```

6.3 Program test results

While testing the program some errors occurred which could not be resolved before this report was finished. They are listed in this section.

1. The adding of two email addresses results in an error. Normally one or several emails are stored in an Amos II vector, displayed on the command line as follows:

   ```
   {"email1","email2"}
   ```

   The result of adding new mails via the PHP script is `{NIL}`. When trying to read the data with PHP function *amos_getstring*, this error message is displayed:

   Fatal error: Error 72: Illegal kind of object in index.php on line 278

   The same problem exists for the telefone numbers that are also stored in an Amos II vector.

2. A similar error appears when changing a person's email address. All email addresses for this person are set to `{NIL}`. The error message is the same like above:

   Fatal error: Error 72: Illegal kind of object in index.php on line 278

   The problem again exists for telefone numbers.

3. A different error occurrs sometimes when trying to display data for a person:

   Fatal error: Error 46: No object numbered in index.php on line 272

   When refreshing the page the error mainly disappears.

7 Amos II tuples and PHP arrays

As studies have come so far, a basic interface has been implemented. All PHP functions (chapter 5) are based on the same idea and structure like the external interface to C. This chapter describes an interesting result that is not implemented or only partly implemented.

Similarities cannot only be seen in the functionality of the Amos II extension and the external interface to C, but also in structures. A connection to Amos II in C is represented as a connection handle, *a_con*. As soon as a connection is established it is stored in an array. The PHP user can access this connection handle by using an index to the storage position. It is valid for all PHP functions that if an index to a connection handle is passed, then the corresponding connection handle will be used for the C function. Scans and tuples are treated the same way.

The conclusion for this implementation is that it is quite an inelegant way, especially for tuples. All tuples contain nothing more than an array with data. As the language PHP also offers arrays to its users it would be better to use arrays in PHP where C uses tuples. Another argument for PHP arrays is storage management. Let's consider the following loop to read a result set:

```
while (!amos_eos($scan)) {
    $tpl = amos_createtuple($con);
    amos_getrow($scan, $tpl);
    /* read data from the tuple by using amos_getXXX */
    ...
    amos_next($scan);
}
```

Let's assume furthermore that scan handle *$scan* holds 10000 tuples. Function *amos_createtuple* would then create a new tuple for each loop, so to say really 10000 tuples. And the variable *$tpl* is only able to reference one of these tuples. To work-around this the user should place the tuple creation in front of the while loop, thus only one tuple will be created. But still it is possible to write such a program. The best solution for this problem are arrays:

```
while (!amos_eos($scan)) {
    $data = array(); // create new array
    amos_getrow($scan, $data);
    /* read data from the tuple by using e.g.  $data[2] */
    ...
    amos_next($scan);
}
```

Function *amos_getrow* will internally allocate a tuple, representing the current row, and copy its complete content into the PHP array *$data*. Finally the internal used tuple can immediately be deallocated again and no memory leaks would stay as PHP has its own storage management that will deallocate the array, if its is not used any more by the script. This would be a great advantage especially for server use.

To realise the connection between Amos II tuples and PHP arrays, two ways need to be considered:

- conversion of an Amos II tuple handle to a PHP array
- conversion of a PHP array to an Amos II tuple handle

7.1 Tuple to array conversion

The first point concerns the PHP function that might return an array, like *amos_getrow*. It is quite easy to create a new array and fill it with Zend macros and functions. A PHP array is stored in a *zval*-container (*new_array* in the example below) which has to be initialized.

```
zval *new_array;
MAKE_STD_ZVAL(new_array);
array_init(new_array);
```

Now this array is ready to fill. Another possibility is to pass an empty array to a PHP function as parameter and initialize it with this function. This second method is applied in PHP function *amos_getrow* which has been extended to

```
amos_getrow(long scan, long tpl, array tofill);
```

Now data from the current row, represented as a tuple handle in C, can be copied into the PHP array by using Zend macros. The copy process is dependent on the data type stored in the tuple. A string must inserted to the PHP array as a string. The same for integers and doubles. An object handle must be transformed to an OID string and added as a string to the array. A tuple can also contain a sequence element. The sequence is represented again by a tuple. Consequently the tuple must be read recursively and PHP arrays must be stored into a PHP array position.

Concerning arrays PHP is very flexible. One array can contain data of any type, including arrays. Figure 18 shows the algorithm that copies all kind of tuple data to a PHP array. In dependence of the Amos II type a PHP function *add_index_xxx* is used to copy the data.

The filled array can be returned by using the code fragment as follows [8].

```
//Make this our $array[0] I do not use add_index_long to
//to show a "by-hand" assignation
zend_hash_index_update(HASH_OF(return_value), 0,
   (void *)&my_long, sizeof(zval *), NULL);

//Returns nothing to satisfy the void prototype
return;
```

7.2 Array to tuple conversion

The second point concerns the PHP functions with an array as parameter, like

```
amos_callfunction(long connection, string function_oid, array argl)
```

instead of

```
amos_callfunction(long connection, string function_oid, long argl_tuple).
```

This approach, to access data from a PHP array and store it into an Amos II tuple, has not been implemented yet. Studies have come so far:

- First of all a header file has to be included:
  ```
  #include "zend_hash.h"
  ```
- An array is retrieved as a *zval*-container with *zval.type = IS_ARRAY*.
- The array itself is stored in *zval.value* as a `HashTable` and can be obtained as follows:
  ```
  HashTable *ht = zval->value.ht;
  ```
- The header file offers some methods for traversing a `HashTable`.
  ```
  zend_hash_move_forward(ht);
  zend_hash_move_backwards(ht);
  zend_hash_get_current_key(ht, str_index, num_index, duplicate);
  zend_hash_get_current_key_type(ht);
  zend_hash_get_current_data(ht, pData);
  zend_hash_internal_pointer_reset(ht);
  zend_hash_internal_pointer_end(ht);
  ```
  First the internal pointer has to be reset. Then by using a method that forwards the pointer, the complete hashtable can be traversed and data copied out of the hashtable.

Now an Amos II tuple can be created internally and filled with data from the `HashTable` by using the *set*-methods from the external tuple interface, like *a_setintelem*, to fill the

```
int fill_array(zval *array, int tpl) {
  int i;
  for (i=0; i<a_getarity(amos_tuple_list[tpl].tuple, FALSE); i++) {
    int type = a_getelemtype(amos_tuple_list[tpl].tuple, i, TRUE);
    if (a_errorflag != 0) {
      AMOS_WARNING;
      return 1;
    }
    switch (type) {
    case INTEGERTYPE: {
      add_index_long(array, i, retrieve_int(tpl, i));
      break;
    }
    case REALTYPE: {
      add_index_double(array, i, retrieve_double(tpl, i));
      break;
    }
    case STRINGTYPE: {
      add_index_string(array, i, retrieve_string(tpl, i), 1);
      break;
    }
    case ARRAYTYPE: {
      int sequence_tuple;
      zval *sequence;
      MAKE_STD_ZVAL(sequence);
      array_init(sequence);
      // retrieve amos vector
      sequence_tuple = create_new_tuple(amos_tuple_list[tpl].con_index);
      if (retrieve_sequence(tpl, i, sequence_tuple) != 0) {
        AMOS_WARNING;
        return 1;
      }
      // recursive call to fill next array
      if (fill_array(sequence, sequence_tuple) != 0) {
        return 1;
      }
      add_index_zval(array, i, sequence);
      break;
    }
    case OIDTYPE: {
      add_index_string(array, i, retrieve_oid(tpl, i), 1);
      break;
    }
    }
  }
  return 0;
}
```

Figure 18: Algorithm for copying tuple data to a PHP array

tuple. Finally this tuple can be used by some Amos II functions as parameter, such as *a_callfunction(c, fid, tuple)*.

An overview as follows lists all necessary PHP functions to Amos II with using PHP arrays instead of Amos II tuples. The amount of PHP functions is strongly reduced as no *set-* and *get-*functions are necessary to fill and read tuples.

- `long amos_connect(string dbname)`
- `void amos_close(long amos_con)`
- `long amos_query(long amos_con, string query)`
- `void amos_closescan(long amos_scan)`
- `bool amos_eos(long amos_scan)`
- `void amos_getrow(long amos_scan, array empty_array)`
- `void amos_next(long amos_scan)`
- `string amos_getfunction(long amos_con, string function_name)`
- `string amos_gettype(long amos_con, string typename)`

-
  ```
  long amos_callfunction(long amos_con, string function_oid,
         array fct_argl)
  ```

-
  ```
  void amos_addfunction(long amos_con, string function_oid,
         array argl, array resl)
  ```

-
  ```
  void amos_setfunction(long amos_con, string function_oid,
         array argl, array resl)
  ```

-
  ```
  void amos_remfunction(long amos_con, string function_oid,
         array argl, array resl)
  ```
- `amos_createobject(long amos_con, string type)`
- `amos_deleteobject(long amos_con, string type_oid)`
- `amos_commit(long amos_con)`
- `amos_rollback(long amos_con)`

This implementation would be a great advantage for the PHP user. He would not have to create a tuple and fill each entry with a seperate PHP function. Instead a PHP array could be created and passed to functions of the Amos II extension. Again for handling result rows, this would be a great advantage. The user retrieves an already filled array, containing all the data of one resulting row, instead of dealing with tuples that have to be read by the *get-*methods of the extension. The last point has been implemented and is doing fine (chapter 5.3.4).

The return of scan handles as arrays has also been considered but rejected again. Filling an array with the complete query result, can result in large memory usage. Result sets normally contain a few columns, what is good for using arrays instead of tuples, but a lot of rows.

8 Conclusion and future work

With this project a complete PHP extension API to access Amos II has been realized. It is possible to establish connections either to one embedded database via a tight connection or to one or several databases where the Amos II system runs as a server. It is a very interesting feature that one client script can open more than one connection to one database. This might be necessary when a PHP based web page is requested by several users, e.g. *n* user requests will open *n* connections to the same database. Each connection handle has its own storage and can be treated seperately.

PHP functions have been implemented to send queries, written im AmosQL, to Amos II peers and to handle returned result scans. At the moment only the tuple interface can be used for this. As outlined in chapter 7 an interesting improvement would be to realise the connection of PHP arrays and Amos II tuples. In addition to this investigations about PHP resources [8] should be made to represent Amos II interface objects such as scans, connections and objects through PHP resources. These features would make the handling of this extension much easier to PHP users that want to access Amos II databases.

Furthermore it is possible to use the *fast-path* interface, which offers a fast way to call Amos II functions, add, set and remove data from a function. Data of a function is defined through its parameters and its result elements. Both items are currently stored in tuples. As soon as these tuples are replaced by PHP arrays it offers an easy to handle way to use the fast-path interface.

Another idea for a future work is to extend the Amos II extension with the *callout* interface. This interface offers the possibility to call external subroutines by using AmosQL statements. To put in concrete terms a PHP user could implement PHP functions, make them known to Amos II and could to call them in queries written in AmosQL. The implemented function should have a signature as follows (similar to the implementation of external C functions):

```
void fn(long ctx, array a);
```

With *ctx* being a reference to an internal Amos II data structure (`a_callcontext`) for managing calls and *a* representing an PHP array that contains the actual arguments and results. First the function parameters are stored and after this the result values.

References

[1] T.Risch, V.Josifovski, and T.Katchaounov: Functional Data Integration in a Distributed Mediator System, in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data, Springer, ISBN 3-540-00375-4, 2003.

[2] Gustav Fahl, Tore Risch: Amos II Introduction. *Tutorial, Dept. of Information Science, Uppsala University, Sweden*, 1999.

[3] Staffan Flodin, Martin Hansson, Vanja Josifovski, Timour Katchaounov, Tore Risch, and Martin Sköld: Amos II Release 6 User's Manual. *UDBL Technical Report, Dept. of Information Science, Uppsala University, Sweden*, March 27, 2004, `http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html`.

[4] Tore Risch: AMOS II Functional Mediators for Information Modelling, Querying, and Integration. *UDBL whitepaper, Dept. of Information Science, Uppsala University, Sweden.* `http://user.it.uu.se/~udbl/amos/amoswhite.html`.

[5] Tore Risch: Amos II External Interfaces. *UDBL Technical Report, Dept. of Information Science, Uppsala University, Sweden*, 2001, `http://user.it.uu.se/~torer/publ/external.pdf`.

[6] Daniel Elin, Tore Risch: Amos II Java Interfaces. *UDBL Technical Report, Dept. of Information Science, Uppsala University, Sweden*, 2000, `http://user.it.uu.se/~torer/publ/javaapi.pdf`.

[7] Kristofer Cassel, Tore Risch. An Object-Oriented Multi-Mediator Browser. *2nd International Workshop on User Interfaces to Data Intensive Systems*, Zurich, Switzerland, 2001.

[8] Stig Sæther Bakken, Alexander Aulbach, Egon Schmid, Jim Winstead, Lars Torben Wilson, Rasmus Lerdorf, Andrei Zmievski, Jouni Ahto: PHP Manual. *PHP Documentation Group*, 12-04-2004, `http://www.php.net/manual/en/`.

[9] George Schlossnagle: Advanced PHP Programming. *Macmillan Computer Pub, ISBN 0672325616*, 2004.

[10] Stephanie Bodoff: Java Servlet Technology. *Sun Microsystems Technical Report*, `http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Servlets.html`.

[11] Stephanie Bodoff: JavaServer Pages Technology. *Sun Microsystems Technical Report*, `http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPIntro.html`.

[12] Govind Seshadri: Understanding JavaServer Pages Model 2 architecture. *JavaWorld Technical Report, 2004*, `http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html`.

[13] Erhard Rahm: Datenbanksysteme 2, Online-Skript, Kapitel 2. *Universität Leipzig, Database Group*, 2003. `http://dbs.uni-leipzig.de/en/skripte/DBS2/inhalt2.html`.

[14] Gio Wiederhold: Mediators in the Architecture of Future Information Systems. *IEEE Computer, 25(3), 38-49*, 1992.

[15] George Shi: Data Integration using Agent based Mediator-Wrapper Architecture. *Tutorial Report*, Dept. of Electrical and Computer Engineering, The University of Calgary, 2002.

[16] Felix Naumann: Qualitätsgesteuerte Anfragebearbeitung für Integrierte Informationssysteme. *it - Information Technology 45(1): 55-58, Lehrstuhl Datenbanken und Informationssysteme, Humboldt Universität zu Berlin*, 2001.

[17] Guido Gerding, Werner Kuhn: A Functional Approach to a Wrapper-Mediator Architecture. *Paper, Universität Münster*, 2003.

[18] Michael Stonebraker, Paul Brown, Dorothy Moore: Object-Relational DBMSs: Tracking the Next Great Wave. *Morgan Kaufmann Publishers, Inc., ISBN 1558604529*, 1999.