Uppsala Master's Theses in
Computing Science No. 288
Examensarbete DVP

Wrapping a B-Tree Storage Manager in an Object Relational Mediator System

Maryam Ladjvardi

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

Abstract

The UDBL group at the department of information technology of Uppsala University in Sweden has developed the extensible Object-Oriented multi database system Amos II (Active Mediator Object System). Amos II is both a database management system (DBMS) of its own and a distributed mediator system. Mediator systems are systems that integrate multiple heterogeneous data sources, providing an integrated global view of the data and providing query facilities on the global view. In order to integrate data sources with different representation formats, the foreign data model must be translated into data model of the mediator system. This is done by defining a wrapper for the data sources. A wrapper consists of interface routines and translation rules. A fast and open data source storage manager, BerkeleyDB, has been developed by the Berkeley database group. It provides a B-tree based external storage representation, similar to what is provided in all commercial relational database products. The goal of this project is to design and implement a BerkeleyDB wrapper for AMOS II, ABKW. The term mediator implies that there are primitives in the system for handling conflicts and similarities between data from different sources. This work only concerns the wrapper part.

Supervisor : Tore Risch

Examinator: Tore Risch

Table of Contents

1. Introduction 4

2. Background 6
   - 2.1 Database Management Systems (DBMS) 6
   - 2.2 Berkeley Database 8

3. Amos II 11
   - 3.1 Amos II Data Model 11
   - 3.2 Query Language 13
   - 3.3 Extencibility 14
     - *3.3.1 Amos II ANSII C Interface* 16
     - *3.3.2 ANSII C Foreign Functions* 16

4. The Amos II Berkeley Wrapper (ABKW) 18
   - 4.1 Architecture 18
   - 4.2 ABKW Foreign Functions Interface 19
     - *4.2.1 Initialization Of The System* 20
     - *4.2.2 Creating Tables* 20
     - *4.2.3 Accessing meta-data* 21
     - *4.2.4 Transactions* 23
     - *4.2.5 Database Updates* 24
     - *4.2.6 Accessing Tables* 25
   - 4.3 Cost Hints 26
   - 4.4 The ABKW Wrapper Interface Generator 27

5. Byte Sort Order Normalization 31
   - 5.1 Encoding Signed Integers 32
   - 5.2 Encoding Floating-Point Numbers 34
   - 5.3 Word Byte Ordering 37

6. Conclusion And Future Work 39

References 41

Appendix: Complete Example 43

1. Introduction

Nowadays many database applications require information from diverse data sources that may use different data representations. The different data representations must be mapped into a single target representation to enable the user to ask queries, transparently, over heterogeneous repositories of information. The *wrapper-mediator* approach allows such transparent queries over data from different data sources. *Wrappers* access data from data sources and *mediators* combine the accessed information under a single schema. The mediator must contain a schema that describes integrated information; in other words the mediator provides an integrated view over the different wrapped data sources that were originally not designed and developed for being a part of a database. Furthermore, the mediator system provides a query language for accessing the data and tries to hide the diversity between the external data sources from the user by creating transparent views on top of them. In order to integrate different data sources, using the wrapper-mediator approach, sufficient wrappers must be defined for data sources to be accessed.

A wrapper is an interface between a mediator and an external data source type that makes the source queryable. It encapsulates the knowledge about the query capabilities of that kind of data source [16]. The wrapper's task is to accept a query from the mediator system, fetch and extract the requested information from the data source, and return the result to the mediator. Converting from one data representation to another is not simply a straight-forward translation between data definition languages. For example, the data types used in one system may not be supported by the other system. Even for identical data types the physical representation of data may create problems: e.g. floating-point representations may differ; the data may, e.g., be represented in *Big Endian* or *Little Endian* form [7]. It is the wrapper's task to map between different data types through feasible functions that map the physical representations.

Amos II is an open, light weight, and extensible database mediator system with its own query language named AmosQL [3, 18], a query language similar to the object-oriented parts of SQL:99 [5]. The aim of a mediator system like Amos II is to integrate data stored in a number of distributed heterogeneous data sources. Data sources may use different data representation techniques, such as a relational database, an object-oriented database, an XML file, etc. Defining wrappers is needed for purpose of making a data source accessible and queryable from the mediator system. Underlying the structure of a database is the data model. In other words a data model is a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints, e.g. the relational model and the object-oriented data model [19]. A task for a wrapper is to transform the data model of a source into the data model of the mediator system, i.e. Amos II [18].

The Amos II mediator system can be extended by defining wrappers to access new kinds of repositories. Then these repositories are accessible from the mediator [4]. The Amos II mediators already contain wrappers for the external data sources Relations databases [2], XML [8], CAD-system [11], and Semantic web [12] wrappers, etc. A wrapper is an embedded subsystem in an Amos II mediator.

A wrapper consists of two subsystems:

- An *interface* that accesses the external data source through a set of foreign functions.
- A *translator* that translates the internal Amos II query predicate representation to data access calls to the data source.

A number of foreign functions must be implemented in the wrapper as interface to the external data source. Once a data source has been wrapped its contents can be queried transparently using AmosQL. The user has no idea from where the data originate but is able to retrieve and update all data by using the common query language. Queries to external sources can be executed with different strategies and thus have different execution costs depending on what strategy is chosen. In order to be efficient and transparent, queries over wrapped sources need to be optimized by the extensible query optimizer inside Amos II [15]. An implementator provides optional *cost functions* that are used by the system for optimizing calls to foreign functions. Cost functions are applied by the query optimizer to compute selectivity and costs. They enable the query optimizer choose the most efficient implementation [16].

The purpose of this project is to design and implement a wrapper between the B-tree storage manager BerkeleyDB [1] and Amos II. Berkeley DB provides a B-tree based external storage representation; similar to what is provided in all commercial relational database products However, unlike commercial database systems, the Berkeley DB data representations is accessed though a low level programmatic interface rather than through the query language SQL. This allows efficient access to the B-tree structures and experimentations with different data representations and query optimization strategies.

The project enables the user transparently to express queries over BerkeleyDB contents using AmosQL. In order to build this wrapper the BerkeleyDB functionality to extract data and the Amos facilities to manipulate the extracted data, using its query language AmosQL, are combined. This combination allows us to design and implement two subsystems of the wrapper, the interface and the translator. In order to perform this work, the following tasks are done:

- The BerkeleyDB system is set up under Windows 2000 and is linked to Amos II.
- A number of foreign functions are defined to access BerkeleyDB data sources. All the foreign functions are implemented through the Amos II interface to the programming language C.
- A simple cost model is defined for Berkeley DB data sources.
- Query update routines are defined for transparent transactional updates of Berkeley DB sources.
- An interface generator is defined that consists of Amos II functions that generate an interface for each table in a specific BerkeleyDB database file.
- The final system is tested and documented using a set of relevant AmosQL statements. The project appendix contains a demonstration of the system functionality.

Furthermore, to efficient execute some queries; *query rewrite rules* should define to transform general database queries into efficient sequences of calls to the primitive Berkeley DB access routines. This is outside the scope of this work.

2. Background

In this section, we first will discuss database management systems in general terms and then we will consider Berkeley DB in particular, and how it fits into the framework we would introduce in the first section.

2.1 Database Management Systems

A database management system, DBMS, consists of a collection of interrelated data and a set of programs to access that data. The data describe, e.g., a particular enterprise. The management of data contains of both the definition of structures for the storage of information and the mechanisms for the manipulation of information.

Data management can be very simple, like just recording configuration in a flat text file. However, in many cases, programs need to store large bodies of data, or structurally complex data. Programmers can do this work quickly and efficiently by using DBMS tools.

The question is, what kind of DBMS can best solve the problems that come up in the application, i.e. knowing the data access and the data management services that the application needs.

All DBMS provide two services:

1. *Data access services* mean the common tasks which are provided by all database systems like, insertion of new data to the database, deletion of data from the database, retrieval of data stored in the database, and modification of data stored in the database. These services are called *data manipulation.* Every DBMS supports a data-manipulation language (*DML*). A DML [19] enables users to access or manipulate data as organized by the appropriate data model. The part of a DML that involves information retrieval is called a *query language*. A *query* is a statement requesting the retrieval of information. The DMLs of modern DBMSs, e.g. SQL, are usually *declarative*, i.e. the user specifies what data are needed without specifying how to search the database to get those data. The database system has to figure out an efficient means of accessing data.

2. *Data management services* are more complicated than data access and different database systems may support different data management services. Most DBMS allow multiple users to update the data simultaneously, called *concurrency*, for the sake of atomic updates and fast response. For this, DBMS usually support *transactions*, which is a collection of operations that performs a single logical update function in a

database application. *Recovery* is another important data management service that is provided by most DBMS. This service means that the database system has fault tolerance i.e. the database system survives application and system crashes.

There are another data management services, for example to provide browsers that show database structure and contents; such services are of less importance here therefore we do not discuss them.

Database systems can be categorized into several *data models*, i.e. ways of representing data that are briefly described as follow:

- *Relational databases* are the most widely used DBMS. A relational database is a collection of data items organized as a set of formally-described tables from which data can be accessed or reassembled in many different ways without having to reorganize the database tables. The tables are sets of rows and columns. You can view the database itself as a set of tables. Relational databases have a very well-known and proven underlying mathematical theory, a simple one (the set theory) that makes possible declarative queries, automatic cost-based query optimization, schema generation from high-level models, and many other features that are now vital for mission-critical information systems development and operations. Relational databases operate on records, which are collection of values of several different data types, including integers, character strings, and others. Operations can be searching for records, updating records and so on. The standard user and application program interface to a relational database is the *Structured Query Language* (SQL). SQL statements are used both for interactive queries for information from a relational database and for gathering data for reports. When creating a relational database, you can define the *domain* of possible values in a data column and further *constraints* that may apply to that data value. For example, a domain of possible customers could allow up to ten possible customer names but be constrained in one table to allowing only three of these customer names to be specifiable [19].

- *Object-based databases* are intended to handle complex data requirements. Current-generation database applications often do not fit the set of assumptions made for older, data-processing-style applications. The object database model has been developed to deal with several of these new type of applications, The changing composition of databases – includes *graphics*, *video*, *sound*, and *text* – require a DBMS which is able to deal with new kinds of data. The object-oriented database is based on the object-oriented-programming language paradigm, which in now in wide use. The model uses object as an abstract representation of a real-world entity that has a unique identity, embedded properties, and the ability to interact with other objects and itself. Attributes are used to describe objects (Also referred to as instance variables in programming languages). An external identifier – the Object ID (OID) is maintained for each object. The OID is assigned by the system when the object is created, and cannot be changed [19].

- *Object-relational databases*, combine features of the relational and object-oriented model. This model provides the rich type system of object-oriented databases, combined with relations as the basis for storage of data [19] along with a complete declarative object-oriented query language to search the database.

2.2 BerkeleyDB storage manager

BerkeleyDB is an open source embedded database library created by Sleepycat software *http://www.sleepycat.com/*. BerkeleyDB is not a full fledged DBMS, but rather a programmatic toolkit that provides embedded database support for both traditional and client/server applications. Such a system is usually called a *storage manager*. Every regular DBMS contains such a storage manager internally. BerkeleyDB is called embedded because it has the option to be directly linked into the address space of the application that uses it, thus, no server to talk to, and no inter-process communication is needed. Once BerkeleyDB is linked into the application, the user has no idea that there is a database present in any way [1].

The BerkeleyDB provides scalable, high performance data management services to applications. It supports a simple function-call API for data access and management. BerkeleyDB provides an ease-to-use interface, allowing programmers to store and retrieve information quickly, simply, and reliably. BerkeleyDB is small without compromising performance and functionality. The database library is scalable which means it can manage databases up to 256 terabytes in size, though itself it is quite compact. It also allows thousands of users operating on the same database simultaneously [1].

BerkeleyDB access methods include B-trees, queues, extended linear hashing, fixed, and variable-length records. In order to identify elements in the database, BerkeleyDB uses key/data pairs, which means that records in BerkeleyDB are (key, value) pairs and all database operations (get, put, delete) are done on the key part of a record and values are simply payload, to be stored with keys and reliably delivered back to the application. BerkeleyDB's access methods are designed store both keys and values as arbitrary byte strings, either fixed-length or variable-length. This simplifies the programmers' job to store native programming language data structures into the database without converting them to a foreign record format [1].

Notice although BerkeleyDB supports key or data items of arbitrary length strings, this is limited by available memory for the largest the key or data item. Specifically, while key and data byte strings may be of essentially unlimited length, any one of them must fit into available memory so that it can be returned to application.

Any of above storage structures can be used to create tables and the mixed operations can be used on the different kinds of tables in a single application. Each of these storage structures is suitable for different kinds of applications, for example, B-trees are well suited for applications that need to find all records with keys between some starting and ending value. Furthermore, the B-trees work well if the application uses keys near each other at the same time, because the tree structure stores the closed keys near one another in storage, therefore the number of disk access is reduced in fetching nearby values [1].

Berkeley DB offers simple data access services to B-trees as follows:

- Insert a record in a table.
- Remove a record from a table.
- Find a record in a table by looking up its key.
- Update a record that has already been found.

- Rang-based searches, i.e. find all records with keys between some starting and ending values.

In contrast to the simple data access services, BerkeleyDB offers significant data management services. It provides full transactional support, database recovery, concurrency, online backups, multi-threaded and multi-process access, etc. All of these services work on all of the storage structures. The library provides strict ACID transaction semantics, by default. ACID denotes *Atomicity*, *Consistency*, *Isolation* and *Durability* properties of the transactions:

- Atomicity: Ensures that each transaction either happens completely, or not at all, and to the outside world, the transaction happens indivisibly, i.e. while a transaction is in progress other processes can not see any of the intermediate states.
- Consistency: Refers to the fact that the transaction does not violate database invariants, i.e. execution of a transaction in isolation (that is, with no other transaction executing concurrency) preserves the consistency of the database.
- Isolation: Ensures concurrent transactions do not interfere with each other. What it means is that if two or more transactions are running at the same time, to each of them and to other processes, the final result looks as though all transactions ran sequentially in some order.
- Durability: Guarantees that, once a transaction complete successfully, the changes are permanent. No failure after the commit can undo the results or cause them to be lost.

In order to isolate concurrent transactions from each other, the two-phase locking technique is used [20]. BerkeleyDB uses write-ahead logging [20] to make sure that committed changes survive application, system, or hardware failures. At the time of starting an application, BerkeleyDB can be asked to run recovery. In this case the database is guaranteed to be consistent and all committed changes are presented when recovery completes. It is the application that determines, at the time of starting, which data management services it will use. The application may choose, for example, fast, single user, and non-transactional B-tree data storage [1]. In this project we use single user transactional data storage to wrap an embedded BerekelyDB system for an Amos II mediator.

With BerkeleyDB terminology a database is represented by a *table*, i.e. tables are databases, *rows* are key/data pairs, and *columns* are application-encapsulated fields within a data item to which BerkeleyDB does not directly provide access. It is possible to create multiple tables within a single physical file called a *database file*. To create or open a database file that includes more than a single database table, a table name must be specified when creating or opening initially the database file. When the first database table is created in a database file, the database file is also created [1].

According to the terminology for a transaction in Berkeley DB, a *transaction* is one or more operations on one or more tables that should be formed a single logical unit of work. These tables may exist in the same database file or in different database files. For example, changes to a set of tables must all be applied to the table(s) or none of them should. It is the task of applications to specify when each transaction starts, what database operations are included in it, and when it ends [1].

In the Berkeley DB system, a database environment can be created. A Berkeley DB environment is an encapsulation of one or more tables, log files, and region files. Region files are shared memory areas that contain information about the database environment. Once the environment has been created, transactions may be started in the environment, and tables may be created and associated within the environment. To create a table, two methods are used. The first method creates a *table handle* with the appropriate environment as an argument, i.e. it allocates a structure for the table. The second method takes the handle, the transaction id, the name of database file, the name of the table, and the type of the table as arguments and creates the table in the database file within the environment [1]. To be able to work against existing tables in the database file, the application must create the handles for each table and open the database within the environment.

BerkeleyDB supports a function-call API for a number of programming languages, including C, C++, Java, Perl, Python, TCL, and PHP. In spite of which data management services are specified by an application, the application uses the same function-call API to fetch and update records. The library is multi environment. It runs on all of the popular operating systems including Windows, all UNIX and Linux variants, and a number of embedded real-time operating systems. It runs on 32-bit and 64 bit systems [1]. In this project we use Windows.

BerkeleyDB is not a relational database system. Relational database systems are semantically rich and offer high-level database access. In contrast to relational databases, BerkeleyDB does not support SQL queries. BerkeleyDB is a high performance transactional library for record storage [1]. It is the role of the BerkeleyDB API to access data. In relational databases, the users by writing simple declarative queries in a high level language can ask questions to the database. The database system knows everything about the data and can carry out the command. This means no programming is required. BerkeleyDB does not have any information on the contents or structure of the values that it stores, i.e. no schema. Therefore, the application needs to know the structure of a key and a value in advance. Furthermore, there is no limit to the data types that can be stored in a BerkeleyDB database. As mentioned earlier, the application never needs to convert its own program data into the data types that BerkeleyDB supports. It can operate on any data type the application uses regardless of how complex the data type is. In order to use BerkeleyDB, the programmer must know the data representation in advance and must write a low-level program to get and store records; in this case, the application can be very fast. Furthermore, it eliminates the overhead of query parsing, optimization, and execution.

BerkeleyDB is not an Object-oriented database system. Object-oriented databases are designed to work well with object-oriented programming languages such Java and C++. Object-oriented databases use similar models as object-oriented programming languages. They operate on the application object by method calls. BerkeleyDB is written entirely in the C programming language. It includes language bindings for C++, Java, and other languages but it never makes method calls on any application object. It does not know what methods are defined on user objects, and cannot see the public or private members of any instance. The key and value part of all records are opaque to BerkeleyDB [1].

Unlike a database server, BerkeleyDB is a library, and runs in the address space of the application that uses it. BerkeleyDB can support more than one application link, and in this way all can use the same database at the same time. The library handles coordination among the applications and makes sure that they do not interfere with one another [1].

3. Amos II mediator system

Amos II is a wrapper-mediator object-oriented, multi-database system. Amos II is a DBMS of its own. The purpose of the Amos II system is to integrate data from many different data sources. A data source can be a conventional database but also text files, data exchange files, WWW pages, programs that collect measurements, or even program that perform computations and other services. Amos II has a functional data model with a relationally complete object-oriented query language, AmosQL [18].

3.1 Amos II data model

The Amos II data model contains three basic constructs: *objects, types* and *functions*. In the data model, everything is an object, including the types and the functions. Each *type* is represented by an *object* of the type 'type' and each *function* is an instance of type 'function' [15, 18].

Every entity is modelled by an object. Object representation is of two kinds, *surrogate* objects and *literal* objects. The surrogate objects have associated explicit object identifiers (OID's) and they are created and deleted by the user of the system. Examples of surrogates object are "real world" entities, such as persons, and meta-objects such as functions. Surrogate objects are removed automatically by a garbage collector when they are no longer referenced from any other object or from external systems [15].

Literal objects are self-describing system maintained objects without explicit OIDs. Examples of literal objects are numbers and strings. Literal objects can also be *collections,* representing collections of other objects. *Vectors* are one of the system-supported collections. A vector is a one-dimensional array of objects. Bags are another kind of collection supported by the system. A *bag* is an unordered set with duplicates [15, 18].

Types describe object structures, i.e. an object can be classified to one or more types, which make the object an *instance* of that type. The set of all instances of a type is called the *extent of the type*. The types are organized in an object-oriented type hierarchy of sub and super types. The type *Object* is the most general type and all other types are subtypes of *Object* [15, 18].

New types are defined as follows:

```
create type Person;

create type Student under Person;
```

Functions are defined on types and used to model the properties of the objects and their relationships. Each function has a *signature* and *implementation*, the signature defines the

types of the arguments and the results of the function. For example the signatures of the functions *name* and *age* of type *Person* can be as follows [15]:

```
name(Person p) -> Charstring nm

age(Person p) -> Integer y
```

The implementation defines how to compute the result of a function given the argument values. For example, *name (p)* obtains the name of the person by accessing the database. AMOS II functions can be *overloaded,* which means they can have the same name with different implementations, called *resolvents* depending on the type(s) and the size of their argument(s). The system has the task of choosing the right implementation to a function call by looking at the types of its arguments [15].

The basic functions can be classified as *stored, derived, foreign*, and *database procedures*, according to their implementations. The implementation of a function is normally non-procedural, i.e. a function only computes values and does not have any side effects. The exception is database procedures, which are special functions having side effects [15, 18].

Stored functions represent properties, attributes of objects stored in the database. Stored functions correspond to attributes in object-oriented databases and tables in relational databases. It has following signature:

```
create function<function name>(<type >) -> <return type> as stored;
```

For example:

```
create function name(Person p) -> Charstring as stored;

create function parent(Person p) -> Person c as stored;
```

Derived functions are defined in terms of other predefined AmosQL functions. They are used to derive new properties that are not explicitly stored in the database. Derived functions cannot have side effects and the query optimiser is applied when they are defined. Derived functions are very similar to side-effect free methods in object-oriented models and views in relational databases. There is an SQL-like *select* statement in AmosQL for defining derived functions [15, 18].

Here is an example of a typical AmosQL derived function:

```
create function sparent (Person p) -> Student s as

       select s where parent(p)=s;
```

This function obtains the parent of a person if the parent is a student.

Foreign functions are implemented through an external programming language and then introduced into the query language [15, 18]. The foreign functions correspond to methods in object-oriented databases and provide access to external storage structures similar to data ‘blades’, ‘cartridges’ or ‘extenders’ in object-relational databases. Foreign functions are used inside wrappers to defined interface to external systems, such as BerkeleyDB.

Foreign functions are often *multi-directional*, i.e. they are invertible that means some unknown arguments can be computed if the result value of the function is known [9]. A *multi-directional foreign function* provides transparent access from AmosQL to special purpose data structures such as internal Amos II metadata representations or user defined storage structures. It can have several implementations depending on the *binding pattern* of its argument and results [9].

The binding pattern is a string of b’s and f’s, indicating which arguments or results in a given implementation are known or unknown, respectively. Multidirectional functions have different implementations depend on different configuration of bound or unbound arguments and results. The programmer has to explicitly assign each binding pattern configuration an implementation. The binding patterns determine the cost of accessing an external data source through an external method. This cost can vary and, to improve the query processing, costing information defined as user functions can be associated with the foreign function. The cost specifications estimate both execution costs in internal cost units and result sizes, *fanouts*, for a given method invocation [15]. There is a simple example of a multidirectional foreign function in section 3.3.2.

To improve degree of executable queries and query optimisation for the system, a multi-directional foreign function can have several associated access path implementations with cost and selectivity functions [9]. With the help of these mechanisms, the programmer is allowed to implement query language operators in an external language such as Lisp, C, or Java and to associate cost and selectivity estimates with different user-defined access paths. The architecture relies on extensible optimisation of such foreign function calls [9]. They are important both for accessing external query processors [9] and for integrating customized data representations from data sources [17].

Database procedures are defined using a procedural sublanguage of AmosQL. They correspond to methods with side effects in object-oriented models.

3.2 AmosQL Queries

The select statement is used to formulate general queries the format of select statement is as follows:

```
select <result> from <type extents> where <condition>
```

For example:

```
select name(p), age(p)
```

```
from Person p
where age(p)>34;
```

The above query will retrieve tuples of the names and ages of the persons in the database who are older than 34 years old.

The semantics of an AmosQL query is in general as follows [15]:

1. Form the Cartesian product of the type extents.
2. Restrict the Cartesian product by the condition.
3. For each possible variable binding to tuple elements in the restricted Cartesian product, evaluate the result expressions to form result tuple.

In order to execute a query efficiently, the system must do extensive query optimisation, otherwise, directly using the above semantic causes very inefficiently execution of the query. This query optimisation transforms the query into an efficient execution strategy. Extending of the query optimizer is performed through multi-directional foreign functions.

Actually, unlike in SQL, AmosQL permits formulation of queries accessing indefinite extents, e.g. all integers, and such queries are not executable at all without query optimisation. For example, the previous query could also have been formulated as:

```
select nm,a
from Person P, Charstring nm, Integer a
where a = age(p) and
      nm = name(p) and
      a > 34;
```

In this case, the Cartesian product of all persons, integers, and strings is infinite so the above query is not executable without query optimisation [15].

3.3 Extensibility

As mentioned before, AMOS II is extensible. Extension of AMOS II is performed by implementing of foreign functions i.e. the interface between an extended AMOS II mediator and an external data source is completely based on foreign functions [15]. Currently there are external interfaces between Amos II and the programming languages C, Lisp and java [16]. In this project, the interface between Amos II and ANSII C is used. In the next section a brief description of the interface is presented.

To map the data model of a data source into the data model of Amos II, the system provides three basic concepts: *mapped types*, *mapped objects*, and *mapped functions*. Since Amos II is an Object-Oriented system each entity is represented by an object. Therefore external data

must also be represented by objects named mapped objects or *proxy objects* that contain no data and are only placeholders without attributes. These objects are instances of a mapped type. In other words mapped types are needed when proxy objects corresponding to external values from some data source are created in a mediator. The instances of mapped types are called mapped objects [2]. In other words, a mapped type is “a type for which the extension is defined in terms of the state of an external database” [2]. In our case, the external database is a B-tree storage manager, BerkeleyDB, and mapped types provide an object-oriented view of data managed by BerkeleyDB.

The attributes of a mapped type are called the *properties* of the mapped type. A mapped function represents functions in other databases and has a mapped type as one of its argument.

A *core cluster function* is a mapped function that defines the instances and primary properties of a mapped type. The core cluster function returns the instances and primary properties as a set of tuples, one for each instance, in some pre-specified order [14].

A mapped type is defined by the following system procedure:

```
create_mapped_type(Charstring typename,
                   Vector keys,
                   Vector columns,
                   Charstring ccfn)
                   -> Mappedtype
```

There the first argument is name of the mapped type, the second one is a vector of the names of the keys identifying each instance of the mapped type, the third argument is a vector of the names of the properties of the mapped type, and the last one is the name of core cluster function.

The core cluster function is an Amos II derived function, which takes no argument and returns a bag of tuples which are the values of their stored properties in some pre-specified order. In other words, it returns a bag of the core properties [4]. The types of the attributes are derived from the types in the result tuple of the core cluster function, which has a signature:

```
<name of mapped type>_CC() -> Bag of <type key, type nonkey,
                                      type nonkey,...>
```

As an example, we assume a table named ‘Person’ and a data source named ‘DB1’. The table ‘Person’ contains three attributes, ssn as primary key, name and age as nonkey attributes, and two records as following:

```
ssn     name      age
1026    sara      25
2048    adam      32
```

If we assume the name of the mapped type is Person_DB1, then the core cluster function has the following signature:

```
Person_DB1_cc() -> Bag of <Integer ssn key, Charstring name, Integer age>
```

And a mapped type is defined as following:

```
create_mapped_type("Person_DB1",
                   {"ssn"},
                   {"ssn","name","age"},
                   "Person_DB1_CC");
```

The result of the call to the core_cluster function, Person_DB1_cc() is as following:

```
<1026,sara,25>
<2048,adam,32>
```

Once the mapped type Person_DB1 is defined, it can be queried as any other type, e.g.:

```
select name(p) from Person_DB1  p where age(p) > 20;
```

The 'key' declaration for element ssn in the core cluster function informs the query optimizer that the attribute is a key. This permits certain kinds of rewrites to significantly improve the performance of the query [2].

3.3.1 Amos II ANSII C interfaces

The external C interfaces are to be used by application systems developers to access Amos II. Hence it is implemented on a rather high level and does not directly give access to Amos II internal primitives [16].

There are two main kinds of external interfaces the *callin* and *callout* interfaces:

- The callin interface is used to call Amos II from a program written in C. In other words the callin API allows the programmer to manage connections to Amos II, send queries to the database, iterate over the result, etc.
- The callout interface is used by foreign AmosQL functions to call external subroutines implemented in C. The callout API allows the programmer to define foreign Amos II functions in C and these foreign functions can then be freely used in database queries. Furthermore, the system permits combination of the two interfaces, the callin interface to be used in the foreign functions, which causes great flexibility and allow them to be used as a form of stored procedures [16].

A *driver* program is needed to call Amos II from C. This program is a C or Java main program that arranges the Amos II environment to be able to call the Amos II system [16].

3.3.2 ANSII C Foreign Functions

AMOS II functions can be defined in C through the callout interface [16]. A foreign AmosQL function written in C contains the following parts:

- The ***C** code* to implement the function. A foreign function is implemented just like any other **C** function with a signature and a body. The signature of a foreign function implementation is as following:

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

  where *cxt* is an internal Amos II data structure for managing the call and *tpl* is a tuple representing actual arguments and results [16].

- A *binding* of the C entry point to a symbol in the Amos II database, i.e. The implementation of the foreign function must be associated with a symbolic name inside Amos II to be able to use it:

  ```
  a_extfunction(char *name, external_predicate fn);
  ```

  where *name* is an identifier for the foreign function and *fn* is a pointer to the **C** function.

- A *definition* of the foreign function in AmosQL, i.e. a function resolvent is created for the foreign function:

  ```
  create function <fn>(<argument declarations>)-> <result declaration>
                                                  as foreign ´<name>´;
  ```

- An optional *cost hint* to estimate the cost of executing the function.

As a very simple example of a multi-directional foreign function, assume we have an external disk-based B-tree table on strings to be accessed from Amos II. We can then implement foreign function *get_string* as follows:

```
create function get_string(Charstring x) -> Charstring st
                                          as foreign "getbtree"
```

Multi-directional foreign functions include declarations of inverse foreign function implementations that means, in our example, the B-tree table can be accessed both by known arguments, keys, and scanned, allowing queries to find all the keys and values stored in the table. Its definition is as follow:

```
create function get_string(Charstring x) -> Charstring y
      as multidirectional ("bf" foreign "getbtree" cost {100,1})
                          ("ff" foreign "scanbtree" cost "scan_cost");
```

Here the foreign function *getbtree* is implemented as a C function and we assume the database contains 100 objects.

The foreign function *scanbtree*, written in **C**, implements scanning of the external B-tree table. The binding patterns, *bf* and *ff*, indicate whether the arguments or result of the function must be bound (b) or free (f) when the external method is called [17].

In the example, the cost specifications are constant for getbtree and computed through the *scan_cost* function for scanning B-trees. The cost function can be implemented as a foreign function in C. The basis for the multi-directional foreign function was developed in [9] where the mechanisms are further described [17].

4. Amos II BerkeleyDB Wrapper

ABKW (Amos-BerkeleyDB-Wrapper) is an interface between Amos II and the external data source BerkeleyDB that enables the user to access and query the external data source by AmosQL queries.

4.1 Architecture

ABKW, according to the principle of an Amos II wrapper, extends the Amos II system by using its C interfaces to define foreign functions that call BerkeleyDB API functions. The architecture of ABKW is shown in figure 4.1 next page.

In the figure, these modules are shown:

- The *ABKW metadata schema* are defined as a set of Amos II foreign functions and derived functions to access meta-data from any BerkeleyDB data source using Amos II foreign functions.
- The *interface generator* consists of Amos II functions that generate an *ABKW table interface* for each table in a specific BerkeleyDB database file with help of an ABKW *metadata schema*.
- An *ABKW table interface* is defined as a number of Amos II functions defining the interface to a specific BerkeleyDB table. It includes a core cluster function definition and a mapped type definition. The table interface functions are used for accessing BerkeleyDB in Amos II queries.
- The *ABKW source interface* is a set of foreign Amos II functions to call the *BerkeleyDB database kernel* through the *BerkeleyDB API*. The database files must use the structure required by ABKW.
- The *BerkeleyDB Kernel* and *BerkeleyDB API* have been provided by Sleepycat [1].
- The *query processor* of Amos II takes a query and transforms it into an efficient execution strategy.
- The *ABKW rewriter* rewrites a query into a semantically equivalent query for an ABKW data source. This module is future work to improve query performance for BerkeleyDB accesses.

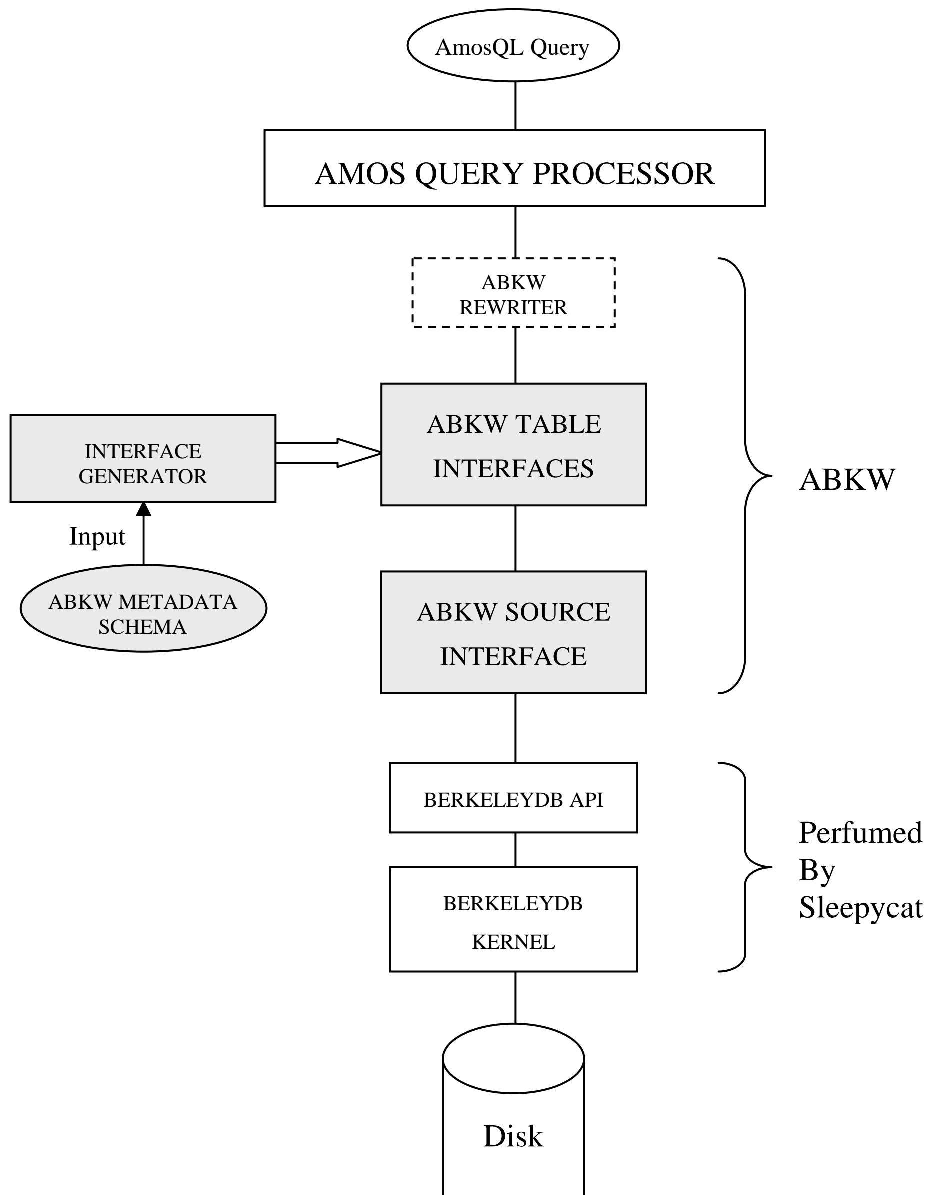

Figure 4.1: Architecture of ABKW. The steps performed in this project are marked grey.
⟹ Shows generation task by the interface generator.

4.2 ABKW Foreign Functions Interface

There are two kinds of foreign functions for accessing ABKW from Amos II:

- Foreign functions that access metadata. The meta-information is stored in the BerekeleyDB database as a B-tree table named “metadata”.
- Foreign functions that are used to access and update the database tables.

The interface functions to BerkeleyDB are all implemented using the foreign function **C** interface of Amos II.

Each BerkeleyDB database file contains several tables implemented as B-trees. A system meta-table in each database files describes all tables in the file. The meta-database management interface allows creation of new database files, and adding new tables to an existing database file, or removing tables from a database.

4.2.1 Initialization of the system

The system must be initialized for every running of the system. This is done by calling the help function *bk_init()*, written in C, in the Amos II driver program. This function has the following signature:

void bk_init()

This function takes no argument, creates a BerkeleyDB environment and starts a transaction in the environment.

Later, during running, *table handles* can be created and opened within the environment and these handles can be enclosed within the current transaction. The transaction is used to do operations on the tables.

4.2.2 Creating tables

The system design permits the system to have a number of B-tree tables in one and the same database file. The size of B-tree tables and files can limit the number of tables in the BerkeleyDB file.

The first thing that must be done is to create a table in the database file. This task is done by a foreign function implemented in C, named *bk_create()* with the following definition:

```
create function bk_create(Charstring dataSource, Charstring tableName,
                          Vector keys, Vector values, Integer unique)
                                         -> Boolean
```

There the two vectors consisting of the name and the type of the keys and the data, respectively. The last argument determines whether the key is unique or not; 1 denotes that the key is unique and 0 denotes that duplicate key values are allowed. It should be noticed that tables with unique key value is of special interest for the query optimizer.

We use a simple example to illustrate our interface. Let us create a table named *'Person'* in a file named *'DB1'* using the foreign AmosQL function bk_create():

```
bk_create("DB1",
         "Person",
          {"ssn","Integer",0},
          {{"name","Charstring",15},{"length","Real",0}},
          1);
```

Where 15 indicates the maximum number of characters which can be used for the attribute *name*. The number of bytes which are used for integers and real numbers are the machine dependent.

Our B-tree table 'Person' stores the properties *ssn, name* and *length* of persons, and ssn is a unique number (indicated by the last argument 1) and can therefore be regarded as primary key value. The data type of ssn is integer, name is string, and length is real number.

In the example, the database file 'DB1' is created if it does not already exists, a '*metadata'* table is created in data file 'DB1', meta-information about the table 'Person' is stored in the metadata table with 'Person' as the key value, and finally the table 'Person' is created.

If the database file 'DB1' already exists, then it also already contains a metadata table. The meta-information about the table 'Person' is stored in the 'metadata' table and the table 'Person' is created.

The foreign function bk_create() must be called by the user of the database once from Amos II directly to create a table. This function creates the database file if it is not already created. It creates there a B-tree table named '*metadata*' to store meta-information about the table and other tables that may be created in the database file later.

In other words, to represent the meta-information about all the tables that exists in a database file, a B-tree table named 'metadata' is created in each ABKW database file. The name of the table, whose meta-information will be stored in the 'metadata' table, is used as key of metadata. There are AmosQL functions defined to access the metadata about the tables (see section 4.2.3)

Furthermore, the table is created and the meta-information about the table is stored in the table named metadata in the database file.

If the database file already exists, then the database file will also contains a metadata table and now by this call the meta-information about the table is stored in the metadata table.

4.2.3 Accessing meta-data

The meta-data table is accessed through the following functions:

```
bk_tables(Charstring datasource) -> Charstring tablenames
```

This function takes a BerkeleyDB database file (data source) as argument and returns the name of the tables stored in the database file. It is a foreign function implemented in C.

```
bk_primkeys(Charstring datasource,Charstring tablename) ->
                                                Charstring keynames
```

This function takes a database file and table name as the arguments and returns the names of the primary keys. It is a foreign function implemented in C.

```
bk_columns(Charstring datasource,Charstring tablename)
                              -> Bag of <columnname, datatype>
```

This function takes a data source and table name as arguments and returns the names and the types of the columns in the table. It is a foreign function implemented in C.

```
bk_columnnames(Charstring src, Charstring tbl) -> Bag of Charstring
```

This function returns the names of the columns and properties of the database *tbl* in the data source *src*. It is implemented as an Amos II derived function with the following implementation:

```
create function bk_columnnames(Charstring src, Charstring tbl)
                                                -> Bag of Charstring
            as select col
               from Charstring tp
               where bk_columns(src,tbl) = <tp, col>;
```

```
bk_nonkeys(Charstring src, Charstring tbl) -> Bag of Charstring col
```

This function returns the names of the non-key properties of a table. It is implemented as an Amos II derived function with the following implementation:

```
create function bk_nonkeys(Charstring src, Charstring tbl)
                  -> Bag of Charstring col
                  as select col
                     where col = bk_columnnames(src,tbl) and
                           notany(select pk from Charstring pk
                                  where pk = bk_primkeys(src,tbl)
                  and
                                        pk = col);
```

The following are examples of the use of the above functions:

```
> bk_tables("DB1");
"Person"

> bk_primkeys("DB1","Person");
"ssn"

> bk_columns("DB1","Person");
<"ssn", "Integer">
<"name", "Charstring">
<"length", "Real">

> bk_columnnames("DB1","Person");
"ssn"
"name"
"length"

> bk_nonkeys("DB1","Person");
"name"
"length"
```

4.2.4 Transactions

As mentioned before, there are foreign functions in ABKW that retrieve information from BerkeleyDB to be used in AmosQL queries to retrieve information there. The user first must connect to the database file. This connection is done by a foreign function *bk_open()* with the following signature:

```
create function bk_open(Charstring datasource) -> Boolean
```

This function takes the name of database file as argument, opens the database file and creates a handle for each table that exists in the database file. This function enables the user to connect to the database file i.e. to work against the tables in the database file.

In other words, this function creates a transient Berkeley handle for each existing table that opens within the environment and the transaction is associated to the handle (The environment and the transaction was created by bk_init() which is described previously). Additionally, this function reads the metadata information about the table from the table 'metadata' and stores them in a global transient array used later during the session. For example, for updating a table, the type of the attributes that are used by the user must match with the metadata of the table. For this checking the metadata information about the table must be available while running of the system. This handle exists until the user commits or aborts the transaction. For details see description of *bk_commit() or bk_rollback()* in the following.

To access these table handles and the information about the tables, respectively, that are created above there is a foreign function *bk_handle()* with the following signature:

```
create function bk_handle(Charstring datasource, Charstring tablename)
                                                        -> Integer
```

This function takes the names of the database file and the table name as arguments. It looks up the content of the global array to check if the table handle to the specified table is there. If it finds the handle table in the array, the actual index of the array is returned as integer, i.e. the array index of the element that contains the handle and the information about the table is returned.

In our example, by calling bk_open("DB1"), all metadata information about the table 'Person' is brought from *metadata* table into the transient global array and a Berkeley handle is created for the table 'Person' and the handle is also stored in the array. bk_handle("DB1","Person") returns the index of the array that contains all metadata about the table 'Person' and its BerkeleyDB handle.

This handle_id used as the argument to other foreign functions during access to the table for simplicity to avoid using the database file and table name again and again. For every new call of bk_handle, the same handle_id always is returned.

After updating a table whether the updates should be committed or rolled back is determined by one of the following foreign functions:

```
bk_commit() -> Boolean
```

The transaction is committed and the transaction handle is closed. According to BerkeleyDB support for transactions, once a transaction is committed or aborted, the transaction handle may not be accessed again [1]. Therefore a new transaction must be started within the environment and the table handles that the committed or aborted transaction is associated to, must also be closed and new handle tables are created enclosed to the new transaction.

For that reason, bk_commit () function immediately starts a new transaction in the BerkeleyDB environment. Furthermore, it closes all table handles associated with the committed transaction and creates new table handles for the tables. These new table handles are associated with the new transaction.

Note that every transaction is also committed automatically when the system is exited if it has not been rolled back.

```
bk_rollback() -> Boolean
```

This function is implemented in the same way as bk_commit () but here the transaction is aborted and a new transaction started.

4.2.5 Database updates

The foreign function *bk_set()* is used to store data in the BerkeleyDB:

```
create function bk_set(Integer handle_id, Vector keys, Vector values)
                                              -> Boolean
```

As an example we populate the table ‘Person’ with four records:

```
> bk_open("DB1");
> set :h=bk_handle("DB1","person");
> bk_set(:h,{1026},{"sara",1.72});
> bk_set(:h,{8234},{"dana",1.40});
> bk_set(:h,{2048},{"adam",1.80});
> bk_set(:h,{2018},{"sam",1.80});
> bk_commit();
```

The records of the tables can be removed from the table by foreign function *bk_del()*

By the following signature:

```
bk_del( Integer handle_id, Vector keys) -> Boolean
```

This function takes handle_id and key as arguments and removes the key/data pairs from the table. If the table supports duplicate, this function removes all the data associated with the key.

For example:

```
> bk_del(:h,{2018});
```

which deletes the key and the data `{"sam",1.80}` associated with the key from the table 'Person'.

The B-tree table can be removed by calling the foreign function bk_remove() that takes the names of database file and the table:

```
bk_remove(Charstring datasoure, Charstring tablename) -> Boolean
```

This function removes the records storing meta_information of the table from *metadata* table and removes the table from the database file. This function can be used at any time during the running of the system to remove an existing table.

4.2.6 Accessing tables

The interface for retrieving data from BerkeleyDB consists of three foreign functions:

- The *exact_get* function retrieves a table row where the primary key is bound i.e. the function takes a key value as argument and returns the associated data. It has the following signature:

  ```
  bk_get(Integer handle_id, Vector keys) -> Bag of Vector
  ```

  In the example:

  ```
  > bk_get(:h,{1026});
   {"sara" 1.72}
  ```

- The *interval_get* function takes two key values, *lower* and *upper* keys as arguments and returns all records whose key/value pairs between these limits:

  ```
  bk_get_interval(Integer handle_id, Vector low_key, Vector up_key)
                                  -> Bag of <Vector,Vector>
  ```

  For example:

  ```
  > bk_get_interval(:h,{1000},{5000});
    <{1026},{"sara",1.72}>
    <{2048},{"adam",1.80}>
  ```

- The *bk_scan* function returns all records stored in the table:

  ```
  bk_scan(Integer handle_id) -> Bag of <Vector, Vector>
  ```

  For example:

  ```
  > bk_scan(:h);
   <{1026},{"sara",1.72}>
  ```

```
<{2048},{"adam",1.80}>
<{8234},{"dana",1.40}>
```

Note that the above access functions are not meant to get called directly by the user but that the wrapper translates parts of a query into these function calls. With the above examples, we only show how they work.

In order to contribute to a higher degree of executable queries and also to improve query optimization for the system, we have defined the following multidirectional foreign function:

```
create function bk_access(Integer handle_id) -> <Vector key, Vector>
       as multidirectional('bff' foreign 'bk_scan' cost bk_cost)
                          ('bbf' foreign 'bk_get' cost{102,1});
```

As described in section 3.3.2 the binding pattern determines which implementation of the multidirectional foreign function to be executed. In our case if the two first arguments are known, the *bk_get()* function is executed; otherwise, the bk_scan() with only the first known argument is executed.

Note that the bk_get() and bk_scan() are resolvents of *BKGet* and *BKScan*, respectively. The *cost hints* defined by the *bk_cost()* function and the constant value *{1002,1}* will be explained later.

In our example, given the ssn number, the name and length of the person can be found quite easily with the function bk_get(). However to find a person by the name or the length, the entire B-tree table must be scanned by using the function bk_scan() follow by a select statement on the intermediate results. The system does this choice automatically based on the cost hints.

Finally, when the user is ready running of the system, the *bk_disconnect()* function must be called. This function commits the active transaction, closes all the table handles, and closes the database environment handle. These tasks disconnect Amos II from the Berkeley database. This function has the following signature:

```
bk_disconnect() -> Boolean
```

4.3 Cost hints

As mentioned before, different implementation of multi-directional foreign functions causes one multi-directional foreign function to have different execution costs. Therefore, the multi-directional function *bk_access()* has different cost dependent on which access functions, bk_get() or bk_scan(), is executed. Each of the two access functions has a cost function. The cost is depending on the actual execution cost of the function and its fanout, i.e. the expected number of records returned. The cost function bk_cost() is defined to calculate the cost of scanning the database since it depends on the size of the table. A constant cost is specified for the bk_get() function. This cost function and the constant cost are applied by the query optimizer to compute selectivity and costs.

The cost function is a foreign function written through Amos II C interface and it has the following signature:

```
bk_cost(Function fn, Vector bpat, Vector args) -> <Integer, Integer>
```

There the first argument is the function for which cost evaluating will be done, in our case it is either bk_get() or bk_scan(). The second argument is binding pattern and the last one is a vector of the names of the database and the source file.

The cost function assumes the cost 2 per record retrieved and this number is multiplied with the numbers of the records in the database and finally the actual calculation cost of the function setup is set to 1000 is added. Consequently, the constant cost is set to 1002.

4.4 The ABKW Wrapper Interface Generator

In order to automatically create a mapped type for each table in a specific BerkeleyDB database file when the database file is connected to, a wrapper interface generator is provided by ABKW project.

The ABKW wrapper generator automatically generates an ABKW wrapper interface for a given BerkeleyDB database file with help of the 'metadata' table in the file. The interface is based on mapped types whose instances are derived from the tables in the database file. The wrapper generator creates mapped types for each table in the source. This requires to automatic generation of the core cluster function of each mapped type based on the metadata for the corresponding data source table. The core cluster function returns the rows of a BerkeleyDB table as a bag of tuples.

Thus the core cluster function for the table 'Person' in data source 'DB1' will have the following name:

Person_DB1_cc()

The following AmosQL stored procedure automatically generates the core cluster function for a given BerkeleyDB table:

```
create function create_bk_ccfn(Charstring src, Charstring tbl) -> Function
   as select
   eval("create function "+bk_typename(src,tbl)+"_cc()
                  -> <"+commalist(bk_column_declarations(src,tbl))+">
                  as select "+commalist(bk_columnnames(src,tbl))+
                  "where bk_access(bk_handle('"+src+"','"+tbl+"')) =
                  <{"+commalist(bk_primkeys(src,tbl))+"},
                  {"+commalist(bk_nonkeys(src,tbl))+"}>;");
```

The core cluster function is generated by calling the system function *eval* to evaluate a string containing an AmosQL statement defining the core cluster function. The components of the generated function definition are obtained by accessing the meta-data table of the wrapped

BerkeleyDB database. The system function '+' is overloaded and does concatenation for strings.

In generating the core cluster function, three derived Amos II functions are used:

Unique mapped type name

```
bk_typename(Charstring src, Charstring tbl) -> Charstring
```

The type names must be unique within each Amos II mediator. Therefore, the name of the mapped type is constructed by concatenating the names of the table and the database file. *bk_typename()* is an AmosQL function forming a unique type name:

```
bk_typename(Charstring src, Charstring tbl) -> Charstring
  as select tbl + "_" + src;
```

For example if a database file 'DB1' contains a table 'Person', the mapped type will be named *Person_DB1.*

Comma lists

```
create function commalist(Bag b) -> Charstring
  as select concatagg(inject(b,","));
```

This function makes comma-delimited string of strings in bag.

For example we assume the following bag of charstring as argument to the commalist function:

"integer ssn"

"charstring name"

The return value of the call to commalist with this argument is:

commalist(bag("Integer ssn","Charstring name");

"integer ssn,charstring name"

From clause

```
create function bk_column_declarations(Charstring src, Charstring tbl)
                                    -> Bag of Charstring
   /* Create bag of Amos II declarations of columns in bk table */
   as select tpn + " " + col
      from Charstring tpn, Charstring col
      where <tpn,col> = bk_columns(src,tbl);
```

This function takes the names of the database file and the table as arguments and creates a bag on Amos II declarations in the from clause of the core cluster function.

For example we assume the database file 'DB1' and the table 'Person' which has two attributes "ssn" of the type integer and "name" of the type charstring. The call

```
bk_column_declarations ("DB1","Person")
```

returns the following strings:

```
"integer ssn"
"charstring name"
```

To test generating of the core cluster function for our example we can generate it like this:

```
create_bk_ccfn("DB1", "Person");
```

Then we can query the generated core cluster function like this:

```
> Person_DB1_cc();
 <1026,"sara",1.72>
 <2048,"adam",1.80>
 <8234,"dana",1.40>
```

In order to completely generate wrapping of a given BerkeleyDB table; a stored procedure in AmosQL has been defined named *bk_generate_wrapper* with the following definition. It has two arguments, the names of the database and data source and creates a mapped type for the database:

```
create function bk_generate_wrapper(Charstring src,
                              Charstring tbl) -> Boolean
 as begin
       declare Charstring ccfn;
       set ccfn = src+ "_" + tbl + "_cc";
       create_bk_ccfn(src, tbl);
       create_mapped_type(bk_typename(src,tbl),  /* Name of mapped type*/
               vectorof(bk_columns(src, tbl)),  /* Mapped attributes*/
          vectorof(bk_primkeys(src, tbl)),   /* Key attributed */
                  ccfn);
       result true;               /* Name of core cluster fn */
 end;
```

In our example, to create the mapped type for the table 'Person', we run this procedure call;

```
bk_generate_wrapper("DB1", "Person");
```

Now we have defined the type Person_DB1 in Amos II with three core properties: ssn, name and length and it can easily be queried as any other types.

Some examples of queries:

```
> select ssn(p) from Person_DB1 p;
 1026
 2048
 8234
```

```
> select ssn(p), name(p) from Person_DB1(p);
 <1026,"sara">
 <2048,"adam">
 <8234,"dana">
> select ssn(p), name(p)
 from Person_DB1 p
 where ssn(p) = 1026;
 <1026,"sara">
```

Note that the result of the above query is determined by the bk_get() function, i.e. will here call the exact_get() implementation.

```
> select ssn(p), name(p)
 from Person_DB1 p
 where ssn(p) > 1000;
 <1026,"sara">
 <2048,"adam">
 <8234,"dana">
```

The result of the above query is computed by the bk_scan() implementation function, i.e. the whole database must be scanned.

To automatically generate wrappers for all tables in a BerkeleyDB data source another stored procedure in AmosQL has been defined named *bk_generate_wrappers*. It takes one argument, the name of the data source, with the following signature:

```
bk_generate_wrappers(Charstring data source) -> Boolean
```

```
create function bk_generate_wrappers(Charstring src) -> Boolean
 as begin
    for each  charstring tbl
     where tbl = bk_tables(src)
      begin
      bk_generate_wrapper(src, tbl);
      end;
    result true;
 end;
```

This function is useful when there are several tables in the data source and will automatically generate all wrappers interfaces for a BerkeleyDB database.

Finally, in order to connect to the database file and automatically generate wrappers for all tables in the database file *bk_connect()* function that has been defined as a stored procedure in AmosQL, is used. It takes the name of the data source as argument and has the following signature:

```
create function bk_connect(Charstring src) -> Boolean
 as begin
      bk_open(src);
      bk_generate_wrappers(src);
 end;
```

In our example:

```
bk_connect("DB1");
```

By this call we have connected to the 'DB1' file and Person_DB1 has been created and we can query the database file.

5 Byte Sort Order Normalization

As mentioned before, the BerkeleyDB stores all information in the database as byte strings, i.e. each byte of key and data pairs is stored as an unsigned character in the memory. This way of storing key causes problems:

- Considering representation of signed integers on a computer, and comparing these numbers with each other as byte strings may lead to wrong results. The negative integer numbers will be larger than the positive numbers and the sorting of the negative numbers will be inverted.
- Considering representation of floating-numbers on a computer, and comparing these numbers with each other as byte string produces incorrect result.
- Considering two different architectures used in designing computers, *Big Endian* and *Little Endian* [7], for handling memory storage, and comparing data as byte strings also produces incorrect results. In a Big Endian system, the most significant value in the sequence is stored at the lowest storage address, in contrast to the Little Edian system where the least significant value in the sequence is stored at the lowest storage address.

  The BerkeleyDB uses a C++ function *DB->set_lorder()*[1] which only sets the byte order for integers keys. This is not enough in our case because in Amos II floating-point type must also be handled and furthermore, Amos II also handles *'compound key'* that consists of more than one column. Therefore ABKW uses its own solution for this problem.

The first and second problems deal with how to convert signed integer numbers and floating-point numbers to/from its corresponding representation in the computer memory. We use two processes: The *encoding process* is done on the key part of the record before inserting the data to the database and the *decoding process* is done after retrieving the data from the database and before sending it to the Amos II kernel. The decoding process is analogous to the encoding one, but in inverse way.

The last problem is handled by converting from Big Endian representation to Little Endian and vice versa when it is needed. In other words, on the Little Endian computer, the converting from Little Endian representation to big Endian one is performed after the

encoding process. The inverse process i.e. the converting from Big Endian representation to Little Endian is done after retrieving the data from the database and before the decoding process. On the Big Endian computer, no conversion is needed.

Note that all above converting are done on the key part of the record because the BerkeleyDB never operates on the value part of a record.

5.1 Encoding signed integers

In this subsection we describe the internal representation of integer numbers, the problem with this representation, and how the problem of encoding and decoding is solved in ABKW.

There are several techniques for representing both positive and negative integers in a computer, sign-and-magnitude, one's complement, and two's complement. The latter is now universally used for representing integers and it is described shortly here:

In *two's complement* format, the highest-order bit is a *sign bit* which makes the quantity negative, and every negative number can be obtained from the corresponding positive value by inverting all the bits and adding one. This is why integers on a 32-bit machine have the range -2^{31} to $2^{31} - 1$. The 32nd bit is used for the sign where a 0 means a positive number or zero and 1 a negative number [13]. As an example, the representation of the number 1 by two's complement format is:

```
00000000    00000000    00000000    00000001
```

The representation of the number -1 is:

```
11111111    11111111    11111111    11111111
```

For comparing the key part of the records stored as byte strings in the database, a lexicographical sort ordering is used by default by BerkeleyDB, with shorter keys collating higher than longer keys. Sort routines are passed pointers to keys as arguments. The routine must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second argument [1]. BerkeleyDB uses a lexicographical comparison to compare fields containing strings as well as numbers. 32-bit numbers are thus regarded as 4-byte binary strings.

For comparing the key part of the records stored as byte strings in the database, the standard C function *memcmp()* is used by ABKW to compare fields containing strings as well as numbers [21]. 32-bit numbers are thus regarded as 4-byte binary strings. The signature of memcmp() is:

```
int memcmp(const void *s1, const void *s2, size_t n);
```

The memcmp() function compares n bytes of two regions of memory. It returns an integer less than, equal to, or greater than zero according to whether *s1* is lexicographically less than, equal to, or greater than *s2*. The parameters are *s1* and *s2*, which points to the first buffer and the second buffer to compare, respectively, and *n* is the number of bytes to compare [21].

If two above numbers are compared by memcmp() function, -1 is larger than 1 because this function treats each byte is treated as an unsigned character, i.e. a byte string. The compared result is thus wrong for integers.

This representation of the signed integer numbers on a computer and using of the lexicographical comparison for comparing these numbers, lead to wrong results. In other words, when we compare integer numbers with each other as unsigned characters, the positive numbers will be smaller than the negative numbers and the sorting of the negative integers will be inverted.

The following example illustrates the inversion of negative numbers:

Consider the representation of two negative numbers by two's complement format [13]:

Representation of -1:

```
11111111    11111111    11111111    11111111
```

and the representation of -256:

```
11111111    11111111    11111111    00000000
```

Comparing these numbers as byte strings gives wrong result, -256 is larger than -1, i.e. the ordering of the negative numbers is inverted.

Therefore, the encoding processes for the signed integer numbers in ABKW is the following:

Encoding integers

A signed integer, *i*, is converted to an *unsigned* integer, *encodei*, by using the system defined maximum unsigned number, *MAXINT*. In this way all the integers will be encoded in the database as unsigned integers:

```
encodei = i + MAXINT
```

This encoding shifts the negative and positive numbers so that all the numbers will be converted to unsigned integer in right sort ordering.

Decoding integers

In order to return an encoded integer from BerkeleyDB to Amos II, the key part of the retrieved record from the database must be decoded. This is done by subtracting MAXINT from the key to get the signed integer:

```
i = encodei – MAXINT
```

5.2 Encoding floating-point numbers

Floating–point numbers must also be encoded before inserting to the database to make them comparable with each other in the correct way using a lexicographical comparison. In the following, we will first describe the representation of the floating-point numbers and then describe the encoding and decoding processes of these numbers.

IEEE 754 floating-point is the most common representation today for real numbers on computers, including Intel-based PC's, Macintoshes, and most UNIX platforms [6]. The floating-point numbers are represented on computers with three basic components: the *sign*, the *exponent*, and the *mantissa*. The sign bit specifies the sign of the real number where 0 denotes a positive number and 1 denotes a negative number. The exponent field represents both positive and negative exponents. To cover the case of negative values, a *bias* is added to the actual exponent in order to get the stored exponent. The exponent base, which is two, is implicit and need not be stored separately. The mantissa is composed of the *fraction* and an implicit leading digit. The mantissa also known as the *significant* represents the precision bits of the number [6].

Floating-point numbers can be either *Single Precision* or *Double Precision*. The single precision representations use 32-bits to represent the floating–point number (*float* number) and in double precision 64-bits are used to represent it (*double* number) [6]. The numbers of bits for each field are shown in figure 5.1 (bit ranges are in square brackets) [6]:

	Sign	Exponent	Fraction	Bias
Single Precision	1[31]	8[30-23]	23[22-00]	127
Double Precision	1[63]	11[62-52]	52[51-00]	1023

Figure 5.1

As an example, we assume the decimal number 23.625 and show how this number is represented as a single precision floating-point number. The binary representation of this number is 10111.101. This number would be represented using scientific notation as $1.0111101 * 2^4$ [6]. The number $1.0111101 * 2^4$ is positive, so the sign field would have a value of 0. As mentioned above, the bias 127 is added to the exponent. Therefore, our exponent would have a decimal value of 131. In binary scientific notation the leading digit cannot be 0 (as decimal scientific notation), it must be 1. So 1 is assumed to always be there and is left out to give us more precision. Thus, the mantissa for our number would be 011101101. Figure 5.2 shows the floating-point representation of this number:

Sign	Exponent	Mantissa
0	1000 0011	0111 0110 1000 0000 0000 000

Figure 5.2

In order to maximize the quantity magnitude of representation of the numbers, floating-point numbers are typically stored in *normalized* form i.e. the mantissa is between 1 and the base [6].

Note that only double precision representation of a floating-point number is used in this project.

Considering the sign of the mantissa and the exponent of the double number, the coordinate line is divided to four ranges:

- The mantissa and the exponent are positive numbers and the double number is greater than 1, i.e. in the interval [1,∞[.
- The mantissa is a positive number and the exponent is the negative number, i.e. the double number is in the interval]0,1[.
- The mantissa and the exponent are negative numbers, i.e. the double number is in the interval]-1,0[.
- The mantissa is a negative number and the exponent is a positive number, i.e. the double number is in]- ∞,-1].

Note that the exponent of the numbers 1.0, -1.0 and 0.0 is zero and the number 0.0 has also zero as mantissa.

These four ranges are shown in the following figure:

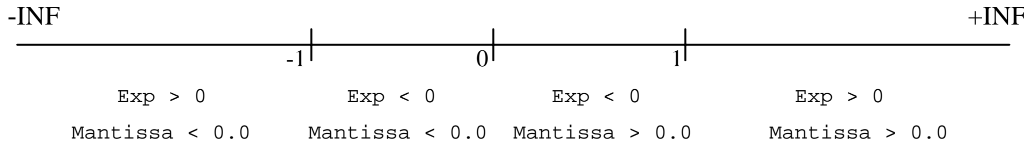

Figure 5.3

To show the problem of comparing the floating-point numbers as unsigned characters we choose four floating-point numbers of the four different intervals as following:

	Decimal numbers	Floating-point numbers		
		Sign	Exponent	Mantissa
1.	4.625	0	1000 0001	0010 1000 0000 0000 0000 000
2.	0.625	0	0111 1110	0100 0000 0000 0000 0000 000
3.	-0.625	1	0111 1110	0100 0000 0000 0000 0000 000
4.	-4.625	1	1000 0001	0010 1000 0000 0000 0000 000

The above example shows that if the floating-point numbers were stored as byte strings in the database and compared as unsigned characters, the results would be wrong. The largest

number, 4.625 will be the smallest one, and the negative numbers will be larger than the positive ones.

The double numbers must be encoded and decoded in a similar way as the integers´ number to enable us to compare them correctly.

To encode the floating-point number, the standard C function *frexp()* is used [22]. This function extracts the mantissa and exponent from a double precision number. It has following signature:

```
double m = frexp(double num, int *exp)
```

The frexp() function thus breaks a floating-point number into a normalised fraction and an integral power of 2. It takes a double number, *num* as the argument and stores the integer exponent in the *int* object pointed to by *exp*. The result value *m,* is a double number with magnitude in the interval [0.5,1] or 0, and num equals m times 2 raised to the power to which exp points. If num is 0, both parts of the result are 0. After breaking the double number into two parts, the mantissa and the exponent, both of the parts are encoded separately.

Assume a floating-point number *num* with double precision and *<exp,m>,* where *exp* is an integer which represents the exponent of *num* and *m* is the mantissa. When encoded by ABKW it is converted to byte sort order normalization by the following rules:

```
-MAXEXP <= exp < MAXEXP      and      0.5 <= |m| < 1
```

where *MAXEXP* is the largest permitted exponent on the computer divided with 4.

If `m < 0.0` and `exp > 0:`

```
encodexp = MAXEXP - exp,  0 <= encodexp < MAXEXP
encodem = m  + 1.0, 0 < encodem <= 0.5
```

If `m < 0.0` and `exp < 0:`

```
encodexp = MAXEXP - exp, MAXEXP <= encodexp < 2*MXEXP
encodem = m + 1.0,  0 < encodem <= 0.5
```

If `m = 0.0:`

```
encodexp = 2 * MAXEXP
encodem = m + 1.0,  encodem = 1.0
```

If `m > 0.0` and `exp < 0:`

```
encodexp = 2*MAXEXP - e, 2*MAXEXP <= encodexp < 3*MAXEXP
encodem = m + 1.0,  1.5 < encodem <= 2.0
```

If `m > 0.0` and `exp > 0:`

```
encodexp = 3*MAXEXP + e, 3*MAXEXP <= encodexp < 4*MAXEXP
encodem = m +1.0,   1.5 < encodem <= 2.0
```

In order to avoid negative mantissa and to fix the correct point 0.0, the mantissa must be added to 1.0

The decoding process is analogous with the encoding process, but in inverse way.

5.3 Word byte ordering

Another problem is the above mentioned Big vs. Little Endian word byte ordering. In BerkeleyDB, the access methods provide no guarantee about the byte ordering of the application data stored in the database, and applications are responsible for maintaining any necessary ordering [1].

When designing computers, there are two different architectures for handling memory storage. They are called *Big Endian* and *Little Endian* [7] and refer to the order in which sequences of bytes are stored in memory. An endianess difference can cause problems if a computer unknowingly tries to read binary data written in the opposite format from a shared memory location or file. So knowing the Endian nature of the computing system is necessary.

The following definitions are more precise:

- *Big Endian* means that the most significant byte of any multibyte data field is stored at the lowest memory address, which is also the address of the larger field.
- *Little Endian* means that the least significant byte of any multibyte data field is stored at the lowest memory address, which is also the address of the larger field.[7]

All processors must be designated at either Big Endian or Little Endian. Intel's 8086 processors and the clones use Little Endian. Sun's SPARC, Motorola's 68K, and the PowerPC families use all Big Endian. Some processors even have a bit in a register that allows the programmers to select the desired Endianess [7].

The Endian nature of the computing system on which a BerkeleyDB application is running can cause a problem when the machine has Little Endian representation and data is stored as byte string. In other words the Little Endian integers and floating-points do not sort correctly when viewed as byte string. The discussion assumes that the numbers are already converted using the actual encoding process.

Why would we care about the byte order representation of integer and double numbers in the database in a Little Endian system? Well, In order to get correct mathematical sorting between numbers. In other words, in a Little Endian system the integer and double numbers have its most significant byte stored at the lowest memory address. Therefore byte strings compared as unsigned characters with the lexicographical comparison are not correctly compared from a mathematical point of view.

In contrast, if the machine has Big Endian representation, everything is all right no matter what type the data has. The data can be stored in the database without any changing of the byte order.

The following example shows two different representations of a number in the two architectures:

Consider the number 1026 (2 raised to the tenth power plus 2) stored in a 4 byte integer:

```
00000000    00000000    00000100    00000010
```

Address	Big Endian representation of 1026	Little Endian representation of 1026
00	00000000	00000010
01	00000000	00000100
02	00000100	00000000
03	00000010	00000000

Note that within both Big Endian and Little Endian byte orders, the bits within each byte are Big Endian.

In the discussion we assume the computing system in which the BerkeleyDB application is run has Little Endian representation.

There are two cases:

- The key part of the records in the database is of the type string; in this case everything is all right. These records are stored in the database as byte string by BerkeleyDB and compared as byte string by ABKW. It works nicely.
- The key part of the records in the database is of the type integer or double. In this case the byte order of the data must be converted from Little Endian presentation to Big Endian one and vice versa. In other words, the key part of the records is stored as Big Endian representation in the BerkeleyDB database by ABKW. To retrieve data from the BerkeleyDB, the key must first be converted to the Big Endian representation. Then the retrieved data, which has Big Endian representation, must be converted to Little Endian. It will then get the original byte order as the one the data had before being inserted into the database. Finally the data is decoded before being sent to Amos II.

For clarification, we assume the binary representation of two numbers 1026 and 2048 in a Little Endian system:

Address:	00	01	02	03
1026	00000010	00000100	00000000	00000000
2048	00000000	00001000	00000000	00000000

If these two numbers are stored as byte strings in the database, the number 2048 is less than 1026 when comparing them with the lexicographical comparison, which is wrong. Therefore, the byte order conversions of numbers are needed to achieve correct results.

In order to address this problem computer system independent, the Endian nature of the system is checked with a small help function *bk_test_endian()* written in **C** that tests if the computer system uses Little Endian. This function takes no argument and returns True if the computer uses Big Endinan, otherwise it returns False. It has the following implementation:

```
bool bk_test_endian(){
int x = 1;
char *p;
p = (char *)&x;
if(p[0]==0)
   return True;
else
   return False;
}
```

By always converting Big Endian numbers to Little Endians the result is correct no matter what Endianess the computer system uses.

6. Conclusion And Future Work

Amos II is an extensible, object-oriented mediator database system. It is concerned to combine the data stored in a number of distributed heterogeneous data sources. In this Thesis, I presented the extension ABKW; I have developed to the Amos II system to wrap tables stored as B-tree in BerkeleyDB data files.

The ABKW system provides a flexible, simple and effective way of accessing and analyzing information from a BerkekelyDB B-tree storage manager through an object-oriented mediator database system. ABKW was implemented using foreign functions through the Amos II interface language programming **C** and BerkeleyDB's B-tree access methods. These foreign functions either called directly by the user or they are directly or indirectly mapped to some other functions.

In this project, metadata was stored in each BerkeleyDB database and a generic ABKW metadata wrapper was defined that automatically generates ABKW data source interfaces. In order to generate an ABKW wrapper the system function *create_mapped_type* was used to define a mapped type and the rest of functions were implemented using Amos II derived functions. This illustrates that wrapping to the system is a simple task.

Two particular problems related to BerkeleyDB's lexicographic byte string representation of numbers had to be solved too.

The first problem is related to the representation of numbers on the computer. With help of encoding and decoding mechanisms, the signed integers' numbers were converted to the unsigned integers so that they could be compared with each other lexicographically in the correct way. The floating-point numbers were also handled so that they were lexicographically comparable.

Furthermore, the problem of Big-Endian vs Little-Endian number representations was addressed making ABKW work independently of the machine on which it is run.

The outcome of the project shows that AmosQL queries can be specified combining data from an AMOS II database with data retrieved from a wrapped BerkeleyDB B-tree storage manager.

What is interesting to be done in the future is an ABKW-rewriter module to handle inequalities in user queries [5]. A foreign function which search the required keys between intervals have already been written to implement rewritten inequality queries.

References

1. *Berkeley DB Reference Guide*, Version 4.2.52, *http://www.sleepycat.com/docs/ref/toc.html*

2. Fahl.G and Risch.T: Query Processing over Object Views of Relational Data. *The VLDB Journal*, Springer, Vol. 6, No. 4, 261-281, 1997, *http://www.it.uu.se/research/group/udbl/html/publ/vldbj97.pdf.*

3. Flodin.S, Hansson.M , Josifovski.V, Katchaounov.T, Risch .T, Sköld. M: *Amos II Release 6 User's Manual*, 2004, *http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html*

4. Gebhardt.Jörn, *Integration of Heterogeneous Data Sources with Limited Query Capabilities*, Linköping Studies in Science and Technology, Master's Thesis No: LiTH-IDA-Ex-99/77, 1999, *http://user.it.uu.se/~udbl/Theses/JornGebhardtMSc.pdf*

5. Hanson.M: Wrapping External Data by Query Transformations, Uppsala Master Theses in Computing Science 243, Dept. of Information Technology, Uppsala, Sweden, 2003, *http://user.it.uu.se/~udbl/publ/AmosCapabilities.pdf*

6. *IEEE Standard 754 Floating Point Numbers, http://stevehollasch.com/cgindex/coding/ieeefloat.html*

7. *Introduction to Endianess*, *http://www.fact-index.com/e/en/endianness.html*

8. Lin.H, Risch.T and Katchanounov.T: Adaptive data mediation over XML data. In special issue on `Web Information Systems Applications` of *Journal of Applied System Studies (JASS) http://www.unipi.gr/jass*, Cambridge International Science Publishing, 3(2), 2002.

9. Litwin.W and Risch.T: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates. *In IEEE Transactions on Knowledge and Data Engineering* 4(6), pp.517-528, 1992, *http://www.it.uu.se/research/group/udbl/html/publ/tkde92.pdf.*

10. Melton.J SQL: 1999 - *Understanding Relational Language Components*, *http://www.service-architecture.com/database/articles/sql1999.html*

11. Nyström.M and Risch.T: Engineering Information Integration using Object-Oriented Mediator Technology, *Software - Practice and Experience* J. *http://www3.interscience.wiley.com/cgi-bin/jhome/1752,* Vol. 34, No. 10, pp 949-975, John Wiley & Sons, Ltd., August 2004.

12. Petrini.J and Risch.T: Processing queries over RDF views of wrapped relational databases, *1st International Workshop on Wrapper Techniques for Legacy Systems, WRAP 2004 http://wwwis.win.tue.nl/wrapper04/*, Delft, Holland, November 2004.

13. *Representation of numbers http://www.swarthmore.edu/NatSci/echeeve1/Ref/BinaryMath/NumSys.html*

14. Risch.T: *AMOS II Functional Mediators for Information Modelling, Querying, and Integration* UDBL, UppsalaUniversity, Sweden, *http://www.dis.uu.se/~udbl/amos/amoswhite.html*

15. Risch.T, Josivofski.V and Katchaunov.T: *AMOS II Concepts*, UDBL, Uppsala University, Sweden, 2000, *http://www.dis.uu.se/~udbl/amos/doc/amos_concepts.html,*

16. Risch.T: *AMOS II External Interfaces*, UDBL, Uppsala University, Sweden, February 2000, *http://user.it.uu.se/~torer/publ/external.pdf*

17. Risch.T, Josifovski.V, and Katchaounov.T: Functional Data Integration in a Distributed Mediator System, in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): *Functional Approach to Computing with Data,* Springer, 2003. *http://user.it.uu.se/%7Etorer/publ/FuncMedPaper.pdf*

18. Risch.T and Josifovski.V: Distributed Data Integration by Object-Oriented Mediator Servers, *Concurrency and Computation: Practice and Experience* 13(11), John Wiley & Sons, September, 2001.

19. Silberschatz.A , Korth.H, and Sudarshan.S: *DATABASE SYSTEM CONCEPTS.,*ISBN 0-07-228363-7, New York McGraw-Hill, 4nd ed., 2002

20. Tanenbaum.A and Steen.M: *DISTRIBUTED SYSTEMS, Principles and Paradigms.* ISBN 0-13-121786-0, Upper Saddle River, NJ: Prentice Hall, 2002.

21. *Unix man pages: memcmp(3), http://bama.ua.edu/cgi-bin/man-cgi?memcmp+3C*

22. Unix man pages: frexp (3), http://fux0r.phathookups.com/unixmanpages/frexp.3.html

Appendix

Here we show a run example of the interface developed in this work, including how to connect to the Berkeley DB from Amos II, how to retrieve information about the database, and some queries that retrieve data from the database.

We first create the Berkeley database 'Person' in the data source 'DB1'. Our B_tree table 'Person' stores the properties ssn of type integer, name of type Charstring and length of type real. ssn is used as primary key value.

```
bk_create("DB1",
                  "Person",
                  {{"ssn","integer",0}},
                  {{"name","string", 15},{"length","real",0}},
                  1);
```

The above function is called just once to create the table 'Person' and store the metadata about the table to the database file..

To access the table handle bk_handle() is called:

```
set :h=bk_handle("DB1","Person");
```

The return value is the handle_id, which is used as the argument for the other queries.

We insert some records to the table:

```
bk_set(:h,{1026},{"sara",1.72});
bk_set(:h,{8234},{"dana",1.38});
bk_set(:h,{2048},{"adam",1.80});
bk_set(:h,{6543},{"sam",1.20});
bk_set(:h,{9873},{"lena",1.65});
bk_set(:h,{1979},{"maria",1.59});
bk_set(:h,{2307},{"david",1.90});
bk_set(:h,{4387},{"tanja",1.62});
bk_set(:h,{6887},{"joe",1.87});
bk_set(:h,{2674},{"lisa",1.43});
bk_set(:h,{6962},{"jan",1.75});
bk_set(:h,{2389},{"frank",1.64});

quit;
```

We have created the table with above records. Now we can connect to the database to query it.

```
bk_connect("DB1");
set :h=bk_handle("DB1","Person");
```

```
bk_access(:h);
<{1026},{"sara", 1.72}>
<{1979},{"maria",1.59}>
<{2048},{"adam",1.80}>
<{2307},{"david",1.90}>
<{2389},{"frank",1.64}>
<{2674},{"lisa",1.43}>
<{4387},{"tanja",1.62}>
<{6543},{"sam",1.20}>
<{6887},{"joe",1.87}>
<{6962},{"jan",1.75}>
<{8234},{"dana",1.40}>
<{9873},{"lena",1.65}>

bk_get_interval(:h,{1000},{4000});
<{1026},{"sara", 1.72}>
<{1979},{"maria",1.59}>
<{2048},{"adam",1.80}>
<{2307},{"david",1.90}>
<{2389},{"frank",1.64}>
<{2674},{"lisa",1.43}>

select ssn(p) from Person_DB1 p;
1026
1979
2048
2307
2389
2674
4387
6543
6887
6962
8234
9873

select ssn(p),name(p) from Person_DB1 p;
<1026,"sara">
<1979,"maria">
<2048,"adam">
<2307,"david">
<2389,"frank">
<2674,"lisa">
<4387,"tanja">
```

```
<6543,"sam">
<6887,"joe">
<6962,"jan">
<8234,"dana">
<9873,"lena">

select ssn(p),name(p)from Person_DB1 p where ssn(p)=1026;  /* Get */
<1026,"sara">

select ssn(p),name(p)from Person_DB1 p where ssn(p)>2640;  /* Scan */
<2674,"lisa">
<4387,"tanja">
<6543,"sam">
<6887,"joe">
<6962,"jan">
<8234,"dana">
<9873,"lena">

select ssn(p),name(p) from Person_DB1 p where length(p)>1.60;
<1026,"sara">
<2048,"adam">
<2307,"david">
<2389,"frank">
<4387,"tanja">
<6887,"joe">
<6962,"jan">
<9873,"lena">

bk_set(:h,{6962},{"jan",1.40});
bk_get(:h,{6962});
{"jan",1.40}
bk_rollback();
bk_get(:h,{6962});
{"jan",1.75}

bk_disconnect();
quit;
```