Uppsala Master's Theses in
Computer Science 301
2005-06-02
ISSN 1100-1836

Wrapping a Scientific Data Management System

Johan Tysklind

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

Supervisor: Tore Risch
Examinator: Tore Risch

Abstract

The UDBL (Uppsala database laboratory) group at Uppsala University has developed an extensible functional multi-database system called Amos II. Amos II can with the help of its interfaces to the external programming languages such as C, Java, and Lisp wrap external data sources. This enables the users to execute AmosQL queries on data that was not originally made for Amos II.

Scientists at CERN in Switzerland have created a framework for C++ that is called ROOT. This framework is mainly created to monitor, visualize, and store data that is created for particle physics. These files that contains huge amounts of data and are not that easily navigated.

The goal of this project is to design and implement a ROOT wrapper that will extend Amos II. With the help of this wrapper the users of ROOT will be able to make queries in AmosQL to search these data.

Preface

I would like to first of all thank Professor Tore Risch, my supervisor, that gave me the chance to do this master thesis and helped me gladly and a lot when I was completely lost. I also would like to thank Ruslan Fomkin that spent several hours to help me setup the whole system on my computer and helped me with all the errors that occurred. I also want to thank the developing team of ROOT at CERN for helping me and answer my questions quickly and accurate. Finally I want to thank friends and family for putting up me for this last semester and also have made my years at Linköping university, Uppsala University, and University of Oklahoma as great as they been.

Västerås, June 2004

Johan Tysklind

Contents

1 Introduction

The largest particle physics center in the world, CERN, is located in Switzerland close to the French border west of Geneva. CERN which stands for Conseil European pour la Recherch Nucleaire, is a joint venture between 20 European member states, the its focus is on particle physics[2].

The CERN laboratory is currently working on finishing the LHC (Large Hadron Collider), which is a particle accelerator. This accelerator will be world's largest accelerator of its kind and will be a ring of 27 kilometers in circumference, and placed 100 meters underground [11]. This accelerator will, when it is finished, produce large amounts of data describing particle events that are produced by proton-proton collisions [9]. But until the particle accelerator is finished scientists around the world, also at Uppsala University, are creating simulations of this accelerator. These simulations produce as well large amounts of data.

The data is right now handled by the ROOT package [1], a C++ package that stores the data in trees (more about the trees later in this report). This package that was developed at CERN is used by all major High Energy and Nuclear laboratories all over the world to monitor, store, and analyze data. The package demands that the scientists has a great knowledge in of C++ programming to be able to navigate in ROOT-managed data to test their theories for how to find particles.

1.1 Objective

This project aims to make the monitoring and analyzing of the data less painful.
The solution is to enable the users to make queries against distributed ROOT data sets. This master's thesis goal is to wrap ROOT to enable high level queries against the data produced by CERN that are stored with the ROOT data management routines. These queries will be carried out in AmosQL, the functional query language for Amos II [6], which is an extensible functional multi-database system. Amos II can be extended so that it can access external data sources, in this case ROOT, with so called wrappers. In this project a wrapper for ROOT has been implemented so that the user can make AmosQL queries that search ROOT files.

The ROOT wrapper uses the interface between Amos II and the programming language C. Currently Amos II has also external interfaces to the programming languages Java and Lisp [5]. These interfaces enable the developer, to make external queries transparent to

the back end data source. The wrapper for ROOT was created in Visual C 6.0 in a Windows environment using the Amos C interface.

1.3 Limitations

The project is limited to work with root data that are stored in ROOT trees, therefore will the not histograms and other objects of ROOT be searchable. Some of the data that was created by CERN are stored with the help of POOL [10] which is a library that extends ROOT. This library will not be included in this project.

1.4 Report overview

The report is dividend into four parts. The first chapter gives the reader an overview over the fundamentals of the project. First database management systems, Amos II, and ROOT are discussed. Second the structure of the wrapper is discussed. Third the structure of the implemented wrapper is discussed. That part of the report will dig deeper into how the wrapper is created. The last part concludes what was done and what can be done in the future to improve this project even more.

Text in italic indicates for the reader that this is a piece of code. Texts that are surrounded by a box are examples to clarify different parts of this project.

2 Background

In this section the background information is discussed, i.e. the topics that are in focus are database management systems, the C++ package ROOT, and finally the database manager system Amos II.

2.1Database Management System

A database management system (DBMS) is a program, or more likely a collection of several programs. A DBMS is designed to manage a database, which is a large collection of data. A database management system is a rather flexible system which can be run locally on a personal computer or on a mainframe; it can handle one user or several users at the same time.

A DBMS contains a set of programs that controls the organization, storage, and retrieval of data (fields, records and files) in a database, as well as security and integrity.

A database system has a data definition language to create schemas, to enable the logic and physical design of user databases. It also has a language that deletes, inserts, manipulates, and retrieves data. This is called a query language that is a language that enables users to access or manipulate data organized by the appropriate data model [7].

There are several different ways that a database system can represent data. These are called Data models:

- The relational data model is a model where the database is represented as tables, which are named with unique names. Each table has a fixed number of columns that are called attributes. All tables have also a set of records, which are the rows in the table. All records in a table must be identified by a unique attribute that is called key; a key can be a single attribute or a set of attributes. It just needs to be a unique combination. All the information about relationships and tables are stored schemas. Operations that can be done on a table are: insert, update, delete, and retrieve data. The interface is a query language which enables a user to request information from the database. The most common query language is Standard Query Language (SQL).

- Object-based databases are made to handle more complex data that the relation model can't handle. This model is based on the object oriented programming paradigm that is used by widespread programming languages like Java and C++ for example. An object based database is an encapsulation of variables, messages, and methods. A variable is an entity in a relationship based database, a piece of data for the object. Messages are chunks of code that is related to the object that will return a property of the object. Finally there are methods that are used to insert and manipulate data within the object. All objects have an ID called an OID that separates the different objects, this ID can't be changed.

- Object-relational databases are a combination of relational database model and the object based database model. An object relational database model extends the relational data model by providing a richer type system including complex data type and object orientation. Object-relational databases provide full support for queries using an object-oriented query language. In this project we enable access to ROOT files from such a query language, called AmosQL [4].

2.2 ROOT

In this section some of the parts of, the particle physics package, ROOT will be explained more into detail. Especially will section will focus on the file system that ROOT uses and the tree structure that ROOT uses to stores most of its data.

2.2.1 Introduction

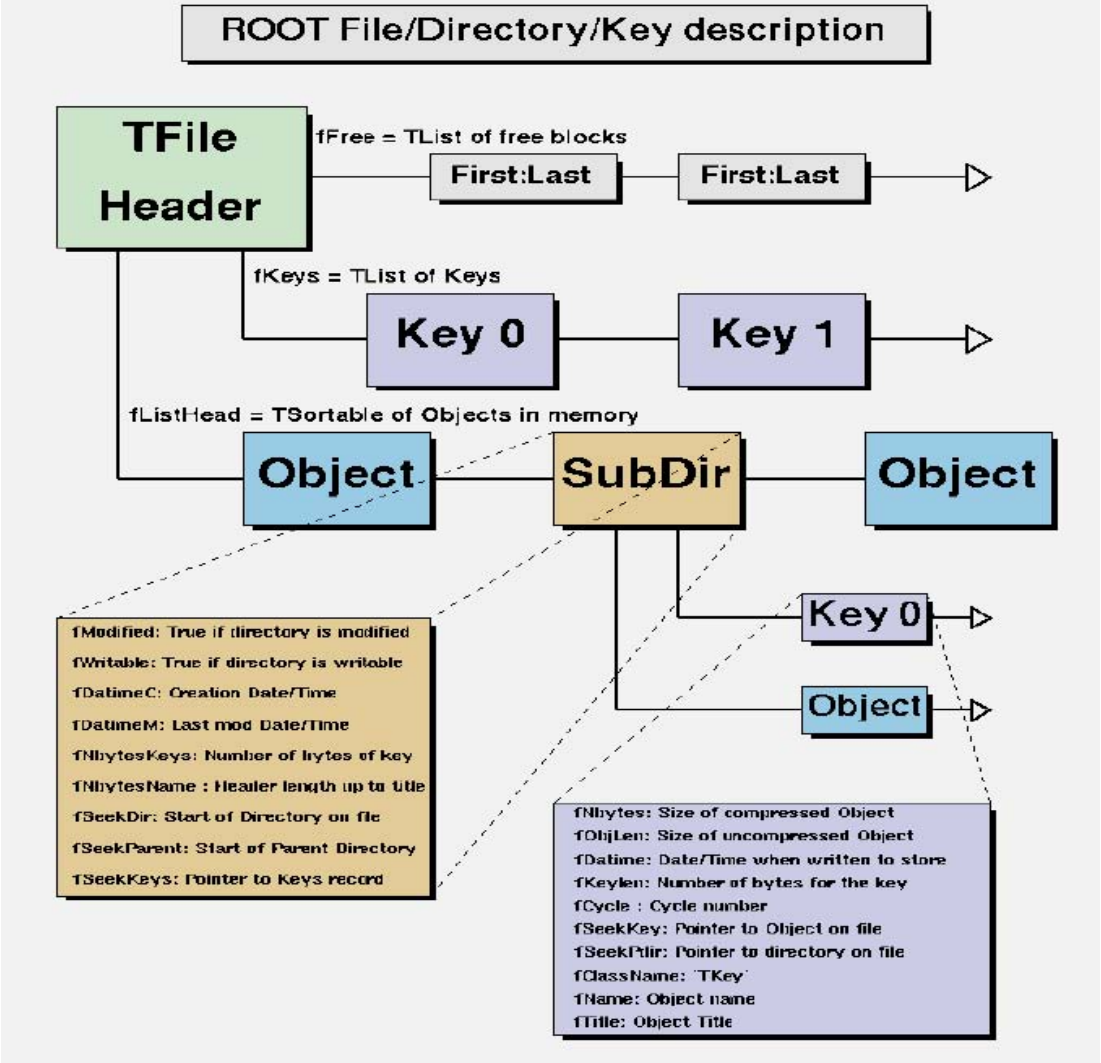
ROOT is a library that was created in Switzerland at the CERN laboratory. The library is an extension to C++ that helps the user to do a lot of scientific calculations, especially for particle physics [1]. ROOT is good for storage, manipulation, and presentation of large amounts of data which is needed in particle physics where the data often are humungous. ROOT is developed by what is called "the bazaar style" [1], which is depending on recommendations and help from the users. The user often wants things from the library that doesn't exist, and if not a person from the development group is doing it, the user can do I himself and add it to the library.

The ROOT library is formed as a framework that helps the user to get hold of functions that other libraries don't provide. In software engineering a framework is like an infrastructure of a city. Advantages with a framework is that you don't have to invent the wheel several of time, several of problems are already solved and the code can be reused. The ROOT library provides functionality such as histograms that can be customized. The code in the framework is rather robust because it's been tested and used by many users. Frameworks make it easier to break a problem into smaller pieces, and also it is easier for users to focus on their problems and not on graphics etc [1].

The ROOT library comes with a C++ command line interpreter and script processor called CINT. An interpreter is a program that takes a program and goes through all instructions and examines it and then executes each statement. An interpreter is good for prototyping, where a compiler is best suited for “compile once run several time”, a interpreter is better when the user has to make a lot of changes and therefore has to compile it many times [1].

2.2.2 File system

A ROOT file is organized much like the UNIX file system. There can be unlimited number of directories and also an unlimited number of levels, i.e. subdirectories. Within these directories the user can save different kinds of data for example he or she can save histograms and trees (we will get back to trees later on). The input and output mechanism in ROOT provides some extra functions like file recovery, where it tries to recover as much as possible if an error occurs [1].



2.2.2 The file system in a ROOT file

Each object in a ROOT file has a header, which is represented as TKey object. TKey objects store information about the object such as name and what class it originates from; it can be a TDirectory, TH1F (1 dimension histogram), TTree, or something similar. In a ROOT file there can be several objects of the same kind.

There are two global variables that help the user to navigate in a ROOT file:

- `gFile` –The pointer to the current opened file.
- gDirecory – The pointer to the current directory.

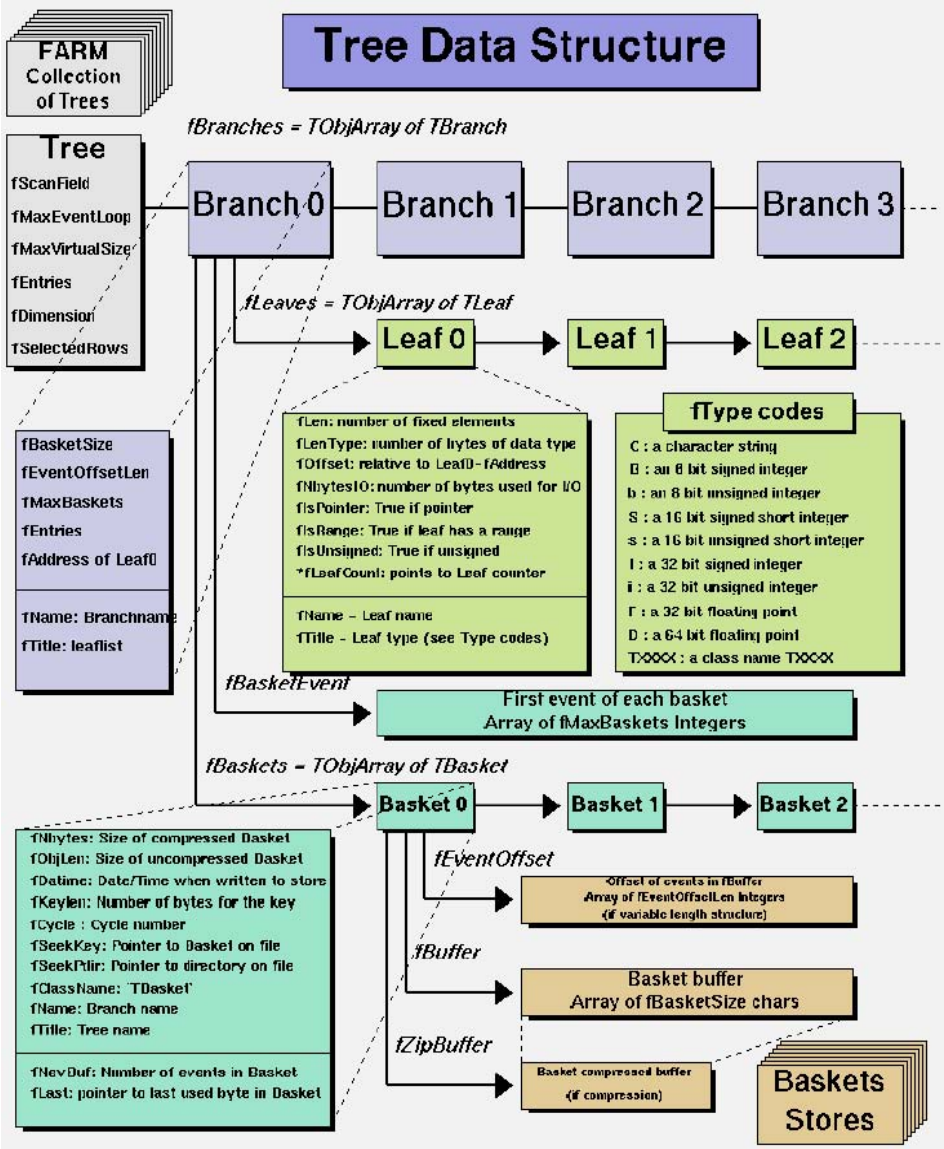
The ROOT library provides a lot of functions to navigate within a ROOT file; they are all very much like the navigation functions in a UNIX operation system. Here are some examples:

- *gDirectory->pwd()* - The *pwd* function will print the current directory.
- *gDirectory->cd()* – This function will move the pointer *gDirectory* to different directories within the ROOT file
- *gDirectory->ls()* – Will list what is in the current directory.

2.2.3 TTree

If the user wants to store large quantities of same class objects, the ROOT library provides two classes: TTree and TNtuple. These classes are both trees that will reduce disk space and increase access time. TNtuple is a class just designed to store floating point numbers, while the TTree class is designed to store any kind of data [1].

The class TBranch is designed to hold the leaves in the tree. The leaves are the ones that hold the data in a tree. Branches are not used like in binary trees where they are used to find a specific object. In ROOT they are used to be a set of leaves, where each leaf holds a set of data objects. This structure can be illustrated like a table in a database, where the tree is the table, the leaf is a column and an event is a row. The branch is a collection of columns that are supposed to be use together, such as coordinates. The branches exist so that the columns easily can be extracted and calculations can be done on these variables. This is also what increases the access speed because it is a lot easier to get a leaf than to get a whole event and then extract a single variable. Unlike relational databases, the rows do not contain atomic values only but also arrays.



2.2.3 The structure of a tree data structure

The user can set the splitting level of the branches. If the splitting level is set at 0 the whole event will be in a leaf that one will be placed into a single branch. So a single branch will hold all columns instead of just the ones that are related. If the splitting level is set to 1, an object data member is assigned a branch. By default the splitting level is set to 99.

Here are some of the most important rules that apply when splitting a branch:

- If a data member is a basic type, it becomes one branch of class TBranchElement.
- A data member can be an array of basic types. In this case, one single branch is created for the array.

- If a data member is an object, the data members of this object are split into branches according to the split level (i.e. split level > 2).

- Base classes are split when the object is split. Base classes are classes that are defined by ROOT.

- Abstract base classes are never split. Abstract base classes are user defined classes.

2.3 AMOS II

AMOS II is a light-weight, Object-Oriented (OO), multi-database system [4]. Amos stands for Active Mediator Object System and is a DBMS. Amos II has a functional data model with an object-oriented query language, AMOSQL. Amos II can get the data from three different sources:

- Data stored in a Amos database

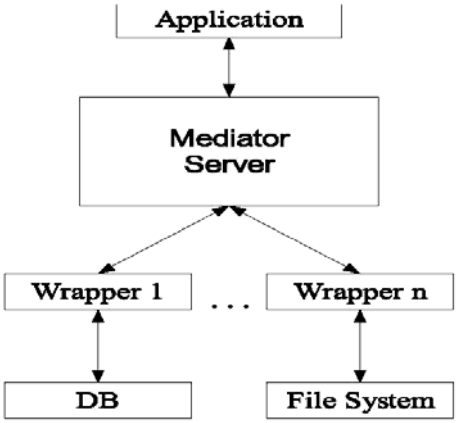
- Wrapped data sources

- Data that is reconciled from other mediator peers.

The second point is extra interesting for this work, because the purpose of AMOS II 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 programs that perform computations and other services [12].

2.3.1 The mediator-wrapper architecture

Today there are a lot of different data sources that are not connected and they do not have the same sort of organization of their data.The mediator-wrapper architecture, allows queries to be transparent over data from different data sources. A mediator wrapper system has one mediator and one or more wrappers, see fig 2.3.1. Wrappers access the data from different data sources and the mediator combines all data into a single schema. A mediator is a system that collects data from different wrapped data sources and combines the result. A mediator can therefore show a view of the wrapped data source that it was not originally made and designed for.



2.3.1 The mediator-wrapper architecture

2.3.2 Amos II data model

The basic concepts of the AMOS II data model are *objects*, *types*, and *functions* [6].

2.3.2.1 Types

All objects in Amos are classified into types making all object an instance of a one or several types. The set of all instances of a type is called the extent of the type [6]. The types are organized in an object-oriented type hierarchy of sub and super types. In the top of the hierarchy is the type Object and under Object are several subtypes.

This is an example how a type is created:

```
create type Person;
create type Student under Person;
```

2.3.2.2 Objects

All entities in an Amos database are modeled as an Object, both the ones that are defined by the users and those who are managed by the system. The objects have two main representations literals and surrogates [6].

The surrogate objects are objects that are created and deleted by users or the system. They are also associated with an object identifier (OID) [6]. Real world objects such as persons or functions can be mention as examples of surrogate objects. A literal object is an object that is system maintained and self-described.

Literal objects have no object identifiers in contrast to the surrogate objects. Literal objects are automatically deleted, when they are not referenced by the database anymore, by the garbage collector. Number and strings are examples of objects that are literals.

Collections can also be literals, collecting other objects. Collections in Amos II are either bags or vectors. A bag is an unordered set with duplicates and a vector is an order preserving collection.

2.3.2.3 Functions

Functions model the semantics and meaning of an object. A function can model properties of an object, computations over objects, and relationships between objects. Each function has a *signature* and an *implementation* [6]. The signature describes the function, what arguments it has, and what it returns. The implementation defines how to use the arguments and what computations need to be done to get the result. An example, a function called name of type person, which signature would then be:

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

The implementation would then be retrieving the name of the person in from the database and return the result.

Depending on the implementation, a function can be classified into one the basic classes: stored, derived, and foreign function [6].

- Stored functions represent properties of objects that are stored in a local database. A stored function is like an attribute in an object oriented database or a table in a entity relationship database.

- Derived functions are functions defined in terms of queries over other Amos II functions. Amos has an SQL-like select statement for defining derived functions.

- Foreign functions are functions implemented in an external programming language. Foreign functions are used to wrap external sources, such as ROOT.

  Foreign functions are often multi directional, this means that they are invertible. That is if the result is known one or several arguments can be computed.

There can be overloaded functions, which means that several of functions can have the same name but different implementations [6]. What separates them is that they have different arguments.

2.3.3 AmosQL queries

Amos II has a query language called AmosQL, queries are formulated with a select statement with the format:

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

A example would be:

```
select name(p) from Person p where age(p) >10;
```

This statement finds all persons that are over ten years of age and returns their names.

In general the semantics of an AMOSQL query is as follows [8]:

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

Queries would be inefficient if they directly used the above semantics. Therefore must the queries be optimized so that there query transforms in to an efficient execution strategy [8].

2.3.4 Extensibility

Amos II is extensible, that is that Amos II can wrap external data sources and make them searchable. To make a data source wrapped, foreign functions needs to be implemented that serves as an interface between Amos II and the external data source. This makes AmosQL queries transparent. The foreign functions can be implemented in one of three external programming languages; Lisp, Java, and C [5]. C is the programming language we are using in this project.

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. A mapped type is “a type for which the extension is defined in terms of the state of an external database” [3]. And a mapped object is a instance of a mapped type. In order to create a mapped type the signature is defined as follows:

```
create_mapped_type(Charstring name, Vector keys, Vector
attrs, Charstring ccfn)-> Mappedtype
```

Where

1. The first argument, name, is the name of the mapped type.
2. Vector Keys is a vector of the keys to this mapped type. In our case there is only one key, ID.
3. Vector attrs are all the attributes that exists in the mapped typed.
4. The last argument is the name of the `core cluster function`.

The core cluster function is a function that returns a set of tuples on for every instance. It is a foreign or derived function that takes no arguments and returns a bag of the core properties. The signature is defined as follows:

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

An example would be to have a database Person with the key SSN and the nonkeys NAME and AGE. The database are filled with these data:

```
Person
SSN         NAME            AGE
1           Adam            30
2           Eve             29
3           Cain            15
4           Abel            10
5           Seth            5
```

Then the core cluster function's signature would look like this:

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

This function would return all the values that are stored in the external database. The mapped type would look as follows:

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

This would make the user able to do transparent queries to the external database. 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 [8].

2.3.5 Foreign Functions

To create a foreign function there are two interfaces that helps to communicate between external systems and Amos II[5]. The two are:

- The Callin interface, this the interface where a programmer can call an Amos II database and retrieve data from the database. This interface is not used in this project.

- The Callout interface, this is the interface that lets AmosQL queries use functions to call external subroutines written in C, Java, or Lisp. The Callout interface is what is used in this project.

In order to create a foreign function there are some steps that needs to be gone trough [5]:

1. C code to implement the function. The signature of the C function implementing a foreign function is as follows:

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

   where the cxt is an internal Amos II data structure for managing the call and the tpl is a tuple that are representing both the argument and the results.

2. A binding of the C implementation to a symbol in the Amos II database. This step is necessary in order to give the function a symbolic name within Amos. The signature is:

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

   The string name is the name that the function will be represented by in Amos. The fn is a pointer to the foreign function.

3. A definition of the foreign function in AmosQL.

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

   In the above AmosQL statement is used, fn is the function name, the argument declaration is the arguments that are passed to the function, the results are the signature of the result that are expected and finally the name of the function that was defined in the binding pattern.

4. An optional cost hint to estimate the cost of executing the function. When there are several alternative implementations of a multi-directional foreign function the cost-based query optimizer needs cost hints that help it choose the most efficient implementation [5].

3 ROOTWrap architecture

In this chapter we will discuss the architecture of ROOTWrap.

3.1 Architecture

The figure 3.1 on the next page shows the architecture of ROOTWrap.

In the figure, these modules are shown:

- The ROOTWrap *table interface* is a number of functions that define the interface to a ROOT tree. This module includes a core cluster function and a mapped type for each ROOT tree.
- The ROOTWrap *source interface* contains foreign functions that retrieve data from the ROOT tree using functions that are defined in ROOT.
- The *ROOT module* is a framework for C++ [1].
- The *query processor* of Amos II takes a query and transforms it into an efficient execution strategy [8].
- The *ROOTWrap rewriter* rewrites a query into a semantically equivalent query for a ROOTWrap data source [8]. This module is future work to improve query performance for a ROOT tree.
- The *interface generator* consists of Amos II functions that generates a interface for each specific tree in that are wrapped in a ROOT file.

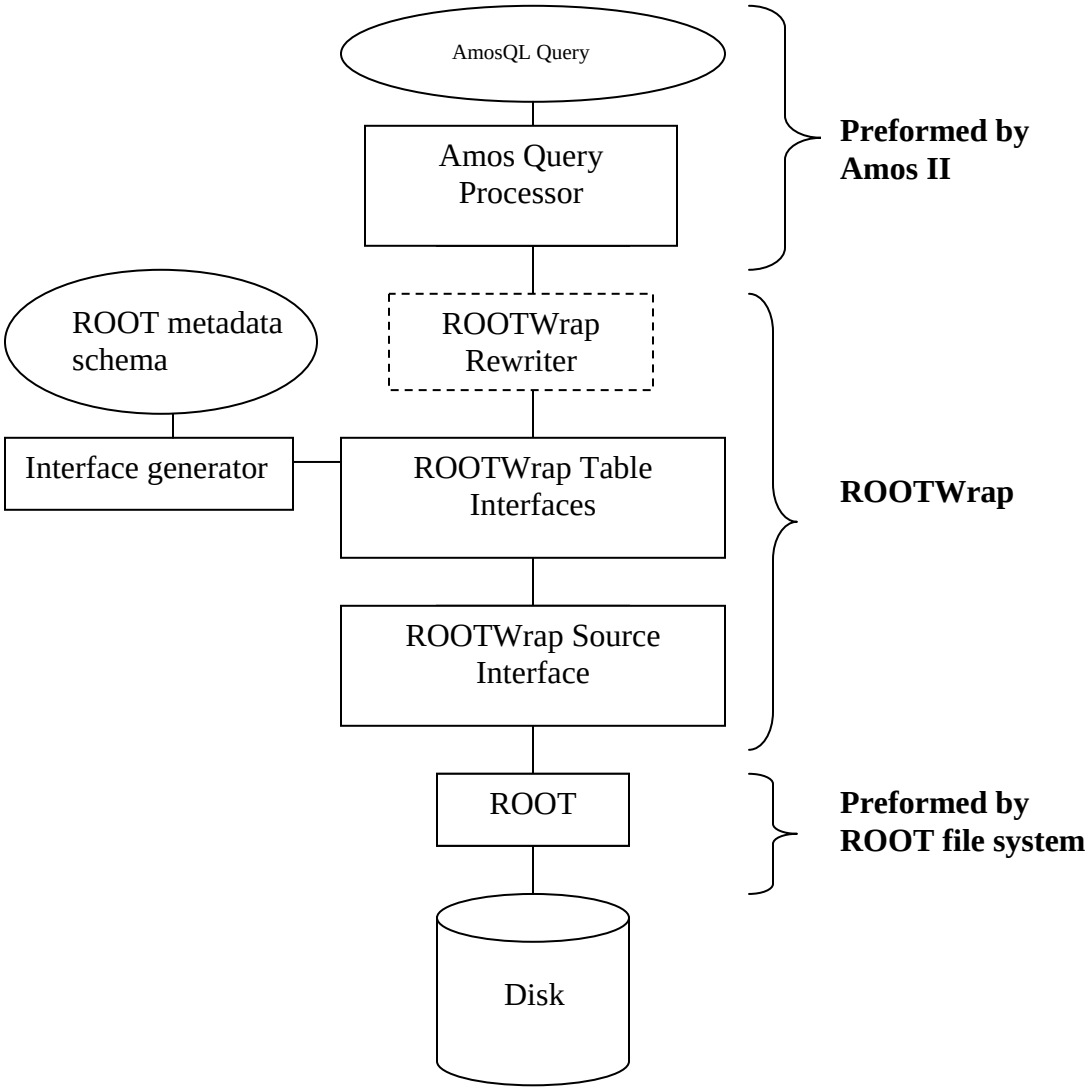


figure 3.1. The architecture of ROOTWrap

3.2 Using ROOTWrap

After setting up the system on a computer, the main function that is needed to be used is `root_generate_wrapper` that genrates a mapped type of a tree and makes the user able to make queries of the data stored in the tree. It thus automatically generates a wrapper interface for a given ROOT data source.

There are a couple of functions that are made for the user to be able to find a certain tree. These are :

- root_dirs, which returns all the directories. This enables the user to search the directories for trees.
- root_files, which returns all trees in a directory.
- root_columns, which returns all the variables in a tree.

When the tree that the user wants is found and `root_generate_wrapper` is called the core cluster function is generated and the user is able to make queries over the data stored in the ROOT tree. The figure 3.2 is an overview of how the functions in ROOTWrap are collaborating.

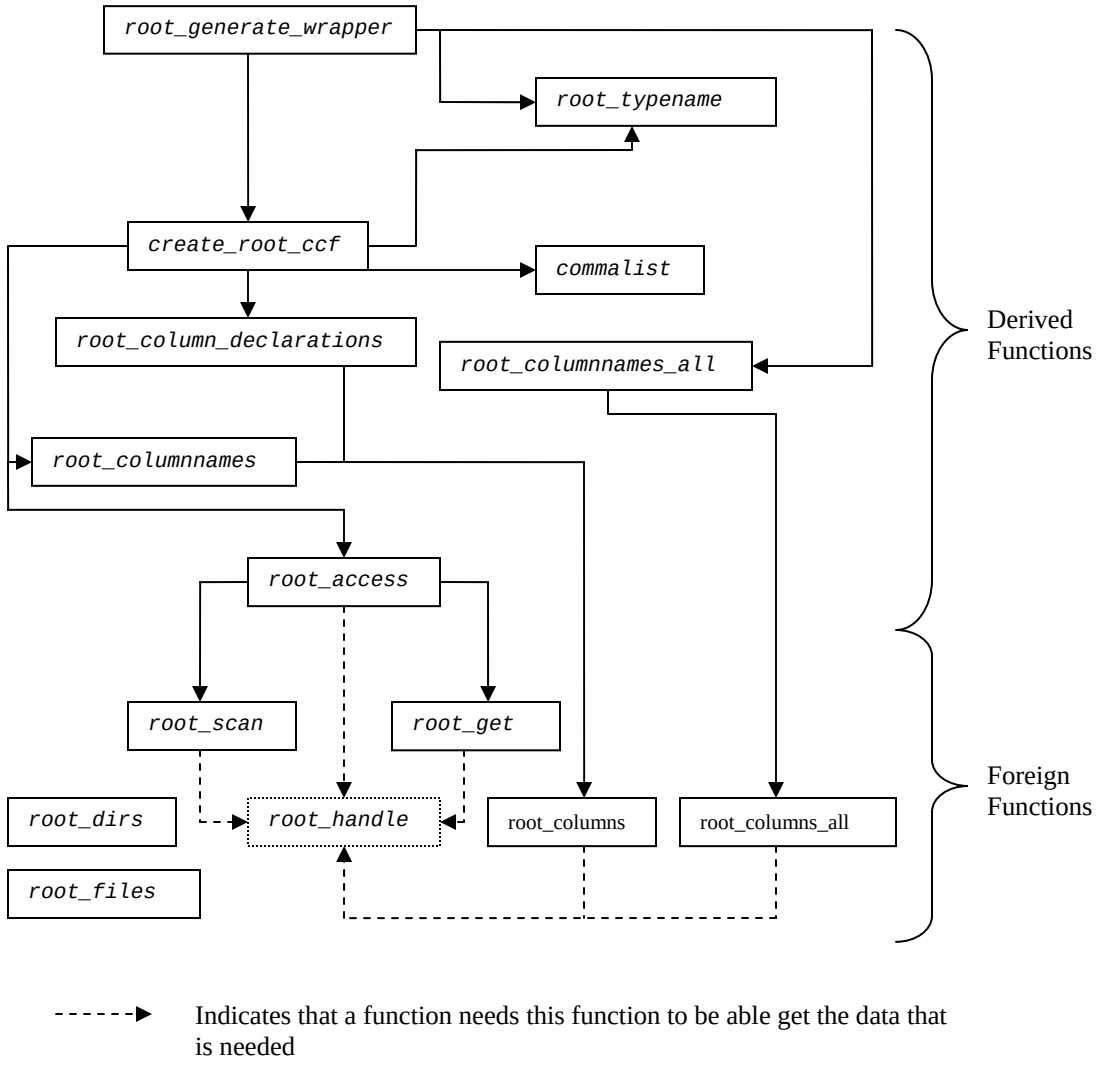


Figure 3.2, the structure of ROOWrap

So called handles are used in some functions. A handle is a number, which is used to retrieve different kind information about a certain tree. Some functions uses this handle as an argument, they gets the number that corresponds to a tree and can from there get the information that is needed, for example what file that is used, what directory the tree is stored in, and what tree we are working on.

4 Implementation of ROOTWrap

We have got two different types of foreign functions in this wrapper:

- Accessing Metadata from a ROOT file. These functions get the file structure of a root file and also the information about the trees that are stored in a ROOTfile.
- Retrieving Data from a ROOT file. These functions either retrieve a single record, a couple of record, or all records in a tree.

There are also some non-foreign functions that are created in AmosQL.

For all the examples we use the ROOT file in figure 4.

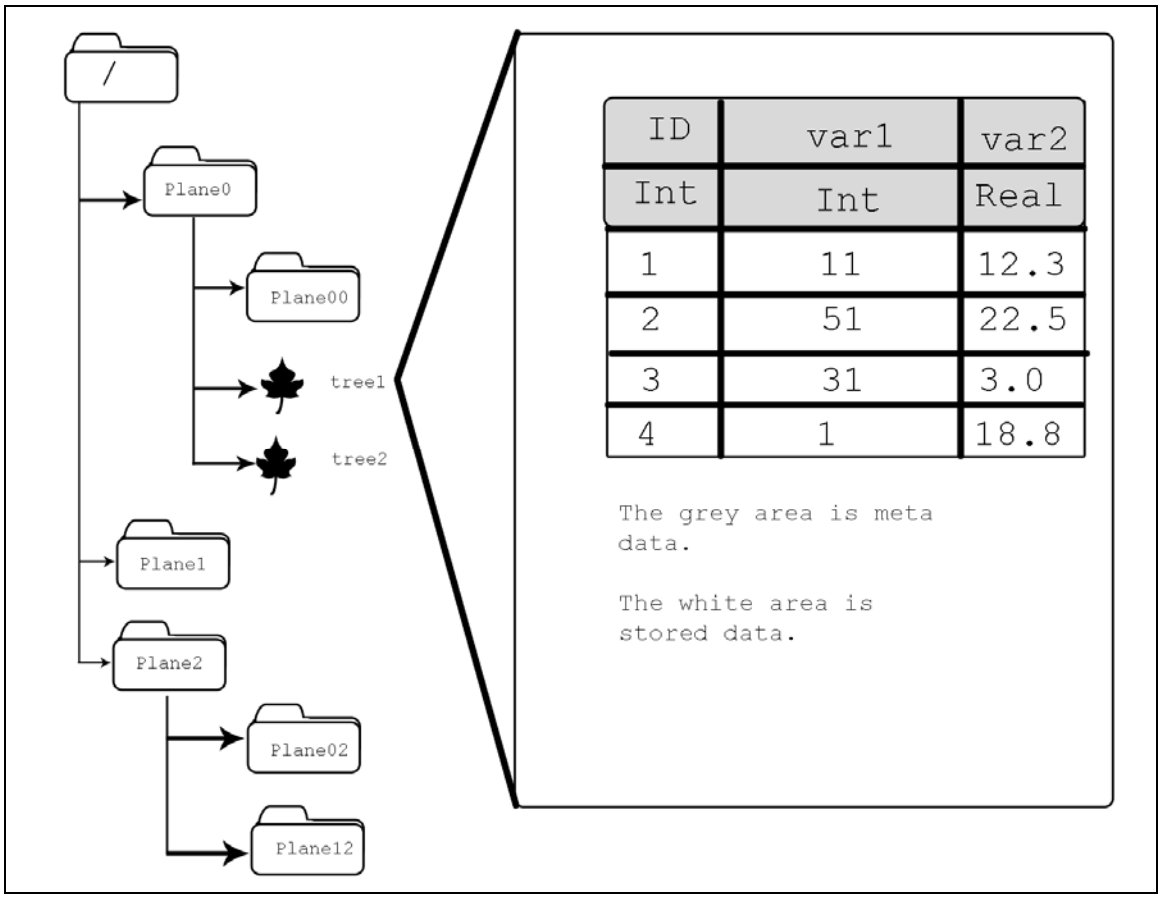


figure 4. An Example of a ROOT file.

4.1 Handling root files

This selection will explain how root_handle makes handling ROOT files both easier and faster.

4.1.1 root_handle

The function `root_handle` handles several ROOT trees and to save data about the trees.

Syntax:

```
root_handle(charstring dataSource, charstring dirName,
charstring tree) -> integer
```

Description:

The `root_handle` function takes a root file, directory and a tree, and stores this information about the tree. The function returns an integer that is the handle that later can be send to other functions so they can retrieve the information about the specified tree fast and easy. Information that gets stored is the access information to a tree (i.e. the root file, directory, and that tree name), the primary key, and all other rows.

Example:

```
> set :h = root_handle("test_file.root","/Plane0","tree1");
```

The variable `:h` will have the number returned by `root_handle`.

4.2 Functions for accessing metadata from a root file

In this section the extraction, with a couple of foreign functions of information about the data stored in all ROOT files, i.e. the Meta data, will be discussed.

4.2.1 root_dirs

This function returns all directories and their subdirectories.

Syntax:

```
root_dirs(charstring dataSource) -> charstring
```

Description:

The function *root_dirs* takes a ROOT file as an argument. The function then goes through the file and returns all the directories and subdirectories within a ROOT file.

Example:

The following examples will use the structure of the file that are shown in figure 1.

```
>root_dirs("test_file.root")
"/"
"/Plane0"
"/Plane0/Plane00"
"/Plane1"
"/Plane2"
"/Plane2/Plane02"
"/Plane2/Plane12"
```

4.2.2 root_ files

This function returns all trees in a directory

Signature:

```
root_files(charstring dataSource, charstring dirName) ->
charstring
```

Description:

There can be several of trees in the same ROOT file and also directories. The function *root_files* takes as a root file and a directory, the function searches the directory and returns all the trees that are stored in that catalog.

Example:

```
> root_files("test_file.root","/Plane0");
"tree1"
"tree2"
```

4.2.3 root_columns

The function *root_columns* returns all variables names and types from a specified tree.

Definition:

```
root_columns(integer handle) -> <charstring
cname,charstring datatype>
```

Description:

The *root_columns* uses the handle that *root_handle* produces to retrieve the information that is needed. The function returns all the names and types of the leaves in the tree, theses leafs will be treated as columns in Amos.

Example (where :h is the same as the one used in root_handle):

```
> root_columns(:h);
<”var1”,” Integer”>
<”var2”,”Real”>
```

4.2.4 root_columns_all

This function is almost the same as *root_columns*, except this function returns a key also.

Signature:

```
root_columns_all(integer handle) -> <charstring cname,
charstring datatype>
```

Description:

The function *root_colums_all* are almost the same as *root_columns*. The difference is that because a tree has no natural key we make one up that is the OID of the event in the tree, i.e. all events have an ID and that is what is used as a key in this project. The function returns all leaves and the key.

Example:

```
> root_columns_all(:h);
<”ID”,”Integer”>
<”var1”,”Integer”>
<”var2”,”Real”>
```

4.3 Functions for getting data from a root file

This section will concentrate on the foreign functions that enables the wrapper to obtain the necessary data.

4.3.1 root_scan

This function returns all events and their data.

Signature:

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

Description:

This function goes through all events in the selected tree and returns all events. The handle is created by `root_handle`. The result is an integer that holds the key of the current event and the vector is populated with the data for that event.

Example:

```
> root_scan(:h);
<1,{11,12.3}>
<2,{51,22.5}>
<3,{31,3.0}>
<4,{1,18.8}>
```

The < … > indicates that it's a vector and this vector holds a key that is an integer and another vector containing the data that was requested.

4.3.2 root_get

This function returns a vector of the data that belongs to a certain event ID.

Signature:

```
root_get(integer handle, integer id) -> Vector
```

Description:

The function, `root_get`, gets a handle that is the handle created by `root_handle` and holds information about the tree that we are going to work on. The function also gets an id that are the id of the event form which our data is supposed to be extracted from. The function will then find the specified event and return the data in a vector, the id will not be returned.

Example:

```
> root_get(:h,3);
{31,3.0}
```

4.3.3 root_get_interval

This function returns one or several events that are stored in a tree.

Signature:

```
root_get_interval(integer handle, integer low, integer
high) -> <Integer,Vector>
```

Description:

The `root_get_interval` function gets a handle produced by `root_handle`. The function also gets two ids; one is the first event that the function should return ant the last one is the last event to be returned.

Example:

```
> root_get_interval(:h,2,3);
<2,{51,22.5}>
<3,{31,3.0}>
```

4.4 The ROOTWrap Wrapper interface generator

An automatic wrapper interface has been created in order to create a mapped type to each tree in a ROOT file. The generator will go through the Meta data information stored in the ROOT file with the foreign functions that were explained above to create mapped types for all trees in the file. To be able to create a mapped type a core cluster function is needed, this and the functions that are need to create a core cluster function will be discussed below.

4.4.1 create_root_ccfn

This derived function creates a core cluster function.

Definition:

```
create function create_root_ccfn(Charstring src,Charstring
dir, Charstring tbl)-> Function
  as select eval(
          "create function "+root_typename(dir,tbl)+"_cc()-
><Integer ID,"+commalist(root_column_declarations(src,
dir,tbl))+">
as select ID, "+commalist(root_columnnames(src,dir,tbl))+"
    where
root_access(root_handle('"+src+"','"+dir+"','"+tbl+"')) =
              <ID,
{"+commalist(root_columnnames(src,dir,tbl))+"}>;");
```

Description:

This function `create_root_ccfn` automatically creates a core cluster function as described above. It creates the core cluster function by calling the eval function that evaluates AmosQL statements. It uses the arguments: ROOT file, directory, and the tree in order to create the core cluster function. The result is concatenated with the system function '+'.

Example:

And if we would run the core cluster function the result would be the following:

```
> tree1_Plane0_cc();
<1,{11,12.3}>
<2,{51,22.5}>
<3,{31,3.0}>
<4,{1,18.8}>
```

4.4.2 root_typename

The function, `root_typename`, concatenates two charstrings to create a unique name.

Definition:

```
create function root_typename(Charstring dir, Charstring
tbl)-> Charstring
      as select tbl + "_" + dir;
```

Description:

In order to create a working core cluster function it needs to have a unique name. This function provides this feature; it concatenates the tree (or table) name with the name of the directory that this tree is placed in. Note that if several of ROOT files are open there can be several core cluster functions that have the same name. This could happen if there are trees with same name that are in the same position in the ROOT file system. If this happens the last made mapped type is going to be the one that is going to be able to be used.

Example:

```
> root_typename(Plane0,tree1);
“tree1_Plane0”
```

4.4.3 commalist

This function changes a bag of strings into a comma based string.

Definition:

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

Description:

The derived functions `root_column_declarations` and `root_columnnames_all` returns bag of strings that need to be concatenated and also there has to be a comma in between the variables to separate them in a select statement.

Example:

```
> commalist(bag(“Integer var1”,”Real var2”);
“Integer var1,Real var2”
```

4.4.4 root_ column_declarations

This derived function converts the vector with variable name and variable type into a string.

Definition:

```
create function root_column_declarations(charstring
src,charstring dir, charstring tbl) -> Bag of Charstring
as
select col+ " " + tpn from Charstring tpn, Charstring
col where <tpn,col> =
root_columns(root_handle(src,dir,tbl));
```

Description:

The foreign functions *root_columns* and *root_columns_all* return bags of vectors that holds two strings, one for the variable name and one for the type, this vector gets transformed into one string that holds the two instances of the vector.

Example:

```
> root_column_declarations
("test_file.root","/Plane0","tree1");
"Integer var1"
"Real var2"
```

Also if we combine the functions *root_coulmn_declarations* and *commalist* the results would be as follows:

```
> commalist(root_column_declarations
("test_file.root","/Plane0","tree1"));
"Integer var1,Real var2"
```

4.4.5 root_columnnames

This derived function returns a bag of strings that contains all variable names in the selected tree.

Definition:

```
create function root_columnnames(Charstring src, charstring
dir, Charstring tbl)
                ->Bag of Charstring
  /* Names of all columns of table */
  as select col
     from Charstring tp
     where root_columns(root_handle(src,dir,tbl)) =
<col,tp>;
```

Description:

This derived function uses *root_columns* in the same way that *root_column_declarations* did, the difference is that this function will only return the variable names. This is used in the select statement in the core cluster function.

Example:

```
root_columnnames("test_file.root","/Plane0","tree1");
"var1"
"var2"
```

4.4.6 root_columnnames_all

This derived function returns a bag of strings that contains all variable names and types, the key is included in this bag.

Definition:

```
create function root_columnnames_all(Charstring src,
charstring dir, Charstring tbl)
                ->Bag of Charstring
  /* Names of all columns of table */
  as select col
     from Charstring tp
     where root_columns_all(root_handle(src,dir,tbl)) =
<col,tp>;
```

Description:

To be able to create a mapped type, you need to list all variables in the wrapped source the key should also be included. In a ROOT tree we don't have a variable that are an obvious key, but instead we are using the OID (Object Identifier) of an event and we are calling it ID. The foreign function *root_columns_all* will return all variables from the tree that have been chosen and also the ID, that is created, *root_column_declarations* will when it gets the result turn these vectors that it gets into bag of strings.

Example:

```
> root_column_declarations
(“test_file.root”,”/Plane0”,”tree1”);
“Integer ID”
“Integer var1”
“Real var2”
```

4.4.7 root_access

This function gets objects from the selected tree.

Definition:

```
create function root_access(integer handle_id )-><integer
key, vector>
as multidirectional
('bff' foreign 'root_scan' cost {1000,1000}) ('bbf' foreign
'root_get' cost {10,1} );
```

Description:

The function `root_access` is used to extract information from tree that is selected in the ROOT file. This function is a so called multidirectional function which is a function that can be inversed [5]. A regular function will give the user a result depending on the arguments that the user passes on to the function, an inversed function will give the argument if it knows the result. The function `root_access` chooses either `root_scan` or `root_get` depending on what the user knows. The binding pattern is what separates them, `‘bff’` means that the `handle_id` is known while the key and vector is unknown and `root_acess` chooses the function `root_scan` that returns all objects for that tree that is associated with the handle. If the `handle_id` and key is known (Binding pattern in this case is `‘bbf’`) we will give the assignment to `root_get` that will extract the object that have the key as OID. This will make the wrapper to run much faster because instead of getting all records and then filter, we just gets the one with the right ID directly from the wrapper.

Examples:

In the case where we don’t know the OID the root_scan will be used, this is an example of a query that would call the scan function:

```
Select var1(t) from tree1_Plane0 t;
```

In the case where you know the OID the of the object and just want that object the root_get function would be called and a query that would call this function could look as follows:

```
Select var2(t) from tree1_Plane0 t where ID(t)=2 ;
```

4.4.8 root_generate_wrapper

The function `root_generate_wrapper` is the function that maps all external data sources.

Definition:

```
create function root_generate_wrapper(Charstring src,
Charstring dir, Charstring tbl) ->Charstring
as begin
declare Charstring ccfn;
 set ccfn = root_typename(dir,tbl)+"_cc";
 create_root_ccfn(src,dir,tbl);
 create_mapped_type(root_typename(dir,tbl),
  vectorof("ID"),
   vectorof(root_columnnames_all(src,dir,tbl)),ccfn);
   result "True";
 end;
```

Description:

In order to create a mapped type there are is an Amos function that does the work, `create_mapped_type`.

The arguments are first the name of the new created mapped type, the second is the key or keys that will separate the different objects, the third argument is all columns in the wrapped data source, and the last argument is the core cluster function.

In our case we are using various derived functions that where defined above to get the arguments that are needed to create the mapped type.

Example:

In our example the call to `create_mapper_type` would look like the following, after that the other derived functions had gathered the information that they assign to get:

```
create_mapped_type(tree1_Plane0,
                   {ID},
                   {“Integer ID,Integer var1,Real var2”},
```

```
                    tree1_Plane0_cc);
```

After this done the mapped type is created and regular AmosQL statements can be executed:

```
Select var1(t) from tree1_Plane0 t;
11
51
31
1

Select var2(t) from tree1_Plane0 t where ID(t)=2 ;
22.5
```

5. Conclusion and Future work

In this section of the report the result is discussed as well as what can be done in the future to improve this project.

5.1 Discussion

Amos II is an extensible functional mediator system, which can execute object-oriented and functional queries over distributed external data sources. In this project Amos II was extended, so that ROOT files can be searched transparently by wrapping the data stored in ROOT files.

The ROOT wrapper made accessing and monitoring data simple and fast to search with the help of several foreign functions. This was made with the C interface that Amos II provides, in a Visual C++ Windows environment.

In order to conclude this work and to make this wrapper to be working a wrapper interface was created. This was made with the help of several derived functions that supported the interface generator `root_generate_wrapper` .

The ROOT wrapper enables the user to do queries in AmosQL and gather the result, updates of ROOT files are not supported..

5.2 Future work

The library POOL [10] is built on top of ROOT. It enables the user to use pointers and links to data sources. A future work could be to make a wrapper for POOL too.

It should also be interesting if the queries could use the `root_get_interval` in the select statements by implementing rewrite rules that transform interval queries into root_get_interval calls. This has been done in the ABKW wrapper [8] and it might be rather simple to transfer that code into this project.

6. References

[1] Brun. R, Rademakers. F, Panacek. S, Antcheva. I, Buskulic. D: *ROOT User Guide 4.04*,CERN, Switwerland, 2004, *ftp://root.cern.ch/root/doc/Users_Guide_4_04.pdf*

[2] *What is CERN?*, *http://public.web.cern.ch/Public/Content/Chapters/AboutCERN/WhatIsCERN/WhatIsCERN-en.html*

[3] 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*.

[4] 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*

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

[6] 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*

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

[8] Ladjvardi M: *Wrapping a B-Tree Storage Manager in an Object Relational Mediator System*, Uppsala Master's Thesis in Computer

Science 288, Dept. of Information Technology, Uppsala, Sweden, 2003, h*ttp://user.it.uu.se/%7Eudbl/Theses/MaryamLadjvardiMSc.pdf*

[9] Fomkin R, Risch T: Parallel Object Query System for Expensive Computations (POQSEC), Uppsala, Sweden, *http://user.it.uu.se/%7Eudbl/poqsec.html*

[10] *POOL - Persistency Framework, http://pool.cern.ch/*

[11] Spotlight on “Angels and Demons” by Dan Brown, *http://public.web.cern.ch/Public/Content/Chapters/Spotlight/SpotlightAandD-en.html*

[12] 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.

7. **Appendix A: Tests and examples**

```
ROOTWrap 2>
root_generate_wrapper("t.root","ATLFAST",
"h51");
[Expanding image to 5047671]
Image moved in PUTHASH
"True"
16.781 s

ROOTWrap 3> select Kfele(p),Nele(p) from
H51_ATLFAST p where Id(p)=3;
<{-11,11},2>
0.437 s

ROOTWrap 4>
root_columnnames_all("t.root","ATLFAST","
h51");
"ID"
"Nele"
"Kfele"
"Pxele"
"Pyele"
"Pzele"
"Eeele"
"Nmuo"
"Kfmuo"
"Pxmuo"
"Pymuo"
"Pzmuo"
"Eemuo"
"Npho"
"Kfpho"
"Pxpho"
"Pypho"
"Pzpho"
"Eepho"
"Nmux"
"Kfmux"
"Pxmux"
"Pymux"
"Pzmux"
"Eemux"
"Njet"
"Kfjet"
"Pxjet"
"Pyjet"
"Pzjet"
"Eejet"
"Ptcalo"
"Ptbjet"
"Ptujet"
"Njetb"
"Kfjetb"
"Pxjetb"
"Pyjetb"
"Pzjetb"
"Eejetb"
"Npart"
"Kppart"
"Kspart"
"Kfpart"
"Kpmoth"
"Kfmoth"
"Pxpart"
"Pypart"
"Pzpart"
"Eepart"
"Isub"
"Jetb"
"Jetc"
"Jettau"
"Pxmiss"
"Pymiss"
"Pxnue"
"Pynue"
0.047 s

ROOTWrap 4> select Nele(p),Id(q) from
h51_atlfast p, h51_atlfast q
       where Id(p)=3 and Nele(p)<Nele(q);
SCAN FUNCTION
<2,586>
<2,715>
<2,1326>
<2,1505>
<2,1567>
<2,1621>
<2,2014>
<2,2247>
<2,2276>
<2,2303>
<2,2357>
<2,2774>
<2,2933>
<2,3053>
<2,3088>
<2,3117>
<2,3140>
<2,3283>
<2,3332>
<2,3533>
<2,3678>
<2,3837>
<2,3859>
<2,4104>
<2,4184>
<2,4305>
<2,4473>
<2,5809>
<2,5966>
<2,6013>
<2,6419>
<2,6637>
<2,6651>
<2,6711>
<2,6721>
```

```
<2,6984>
<2,7106>
<2,7199>
<2,7485>
<2,7723>
<2,7785>
<2,7926>
<2,8175>
<2,8325>
<2,8340>
<2,8427>
<2,9301>
<2,9343>
<2,9461>
<2,9582>
<2,9667>
<2,10282>
<2,10372>
<2,10467>
<2,10602>
<2,10658>
<2,10663>
<2,10792>
<2,11381>
<2,11400>
<2,11627>
<2,11750>
<2,12089>
<2,12266>
<2,12279>
<2,12431>
<2,12497>
<2,12718>
<2,12900>
<2,13570>
<2,13824>
<2,13867>
<2,13919>
<2,13948>
<2,13984>
<2,14055>
<2,14093>
<2,14175>
<2,14464>
<2,14501>
<2,14729>
<2,14747>
<2,14764>
<2,14792>
<2,14835>
<2,14856>
<2,15192>
<2,15552>
<2,15579>
<2,15717>
<2,15815>
<2,15861>
<2,16058>
<2,16115>
<2,16146>
<2,16361>
<2,16364>
<2,16668>
<2,16818>
<2,17011>
<2,17051>
<2,17279>
<2,17324>
<2,17413>
<2,17432>
<2,17562>
<2,17764>
<2,17808>
<2,17834>
<2,17837>
<2,17846>
<2,18125>
<2,18313>
<2,18880>
<2,19270>
<2,19475>
<2,19620>
<2,19921>
<2,19946>
<2,20114>
<2,20143>
<2,20424>
<2,20595>
<2,20659>
<2,20908>
<2,20934>
<2,20951>
<2,20981>
<2,21030>
<2,21230>
<2,21265>
<2,21522>
<2,21591>
<2,21623>
<2,21823>
<2,22050>
<2,22134>
<2,22277>
<2,22327>
<2,22584>
<2,22585>
<2,22599>
<2,23069>
<2,23088>
<2,23171>
<2,23195>
<2,23348>
<2,23429>
<2,23519>
<2,23524>
<2,23534>
<2,23720>
<2,23817>
<2,23850>
<2,24279>
<2,24355>
<2,24439>
<2,24446>
<2,24455>
<2,24807>
5.234 s
```