The AIPS++ Table Data System

The present state of the AIPS++ Table Data system, 15 October 2006
DRAFT Version

G. van Diepen and A. Farris

Last modified: $Date: 2003/07/16 04:19:18 $ By: $Author: aips2adm $

Introduction

In analysing some of the major applications of AIPS++ and attempting to address a wide range of requirements, it is clear that the underlying data system will play a key role; not only must it cope efficiently with the requirements of applications, but also the with the fact that users/application programmers wish to have flexible access to their data to implement new algorithms and techniques. In addition, flexible data access and selection is vital to enable the rapid development of applications for visualising and interacting with data.

The AIPS++ Table Data System is strongly based on the table model outlined by Allen Farris. This document describes the design and implementation of the table system. It is divided into 3 parts:

1. The first chapter describes the table model and the relation with FITS and is intended as a background reading.
2. The second chapter gives an overview of the classes in the table system. The following chapters describe how these classes can be used to store and retrieve data in/from a table. Appendix A [Examples], page 48 contains some examples to make it a bit more clear.
3. The last chapter gives a more detailed description of the implementation the main table classes. It is intended for the programmers who derive other classes from them to implement so-called virtual tables. However, especially the introduction is also very instructive for the user of the table classes.

1 Table Model

1.1 Background and Goals

The material in this chapter is derived almost entirely from a draft document on "A Database Manager for Astronomical Data Analysis" written by Allen Farris. It describes a new database model conceived as an extension to the traditional relational model[1]. The functions of a database manager to support the model will also be discussed. The principal goal of the model and the database manager is to support astronomical data analysis, which must deal with data characterized by complex internal structures and the need to manipulate multi-dimensional arrays. Because these features of astronomical data are common to scientific data in general, the model presented here is applicable to many areas of science, not just astronomy. However, the focus of this paper will be on astronomical data.

While the model to be presented is an extension to the relational database model, we will represent the model using the concepts surrounding FITS binary tables[2]. FITS is well known in astronomy and the traditional relational database table can be viewed as a special case of a FITS binary table. The use of FITS is purely a convenience within an astronomical context; the model itself is not dependent on FITS. It is also the intention of this paper to merely use the ideas surrounding FITS and to build upon them. Changes to the existing FITS standard within astronomy are not being proposed.

The traditional relational database model has two features that limit its applicability to scientific data. First, an item of data in a column of a table must be atomic; it must have no internal structure. A consequence of this restriction is that relational databases are unable to deal with arrays of data items. Second, an item of data in a column of a table must not have any direct or implied linkages to other items of data or data aggregates. This restriction makes it difficult to model complex relationships between collections of data. While these restrictions may make it easy to define a mathematically complete set of data manipulation operations, they are simply intolerable in a scientific data-handling context. Multi-dimensional arrays are frequently the most natural modes in which to discuss and think about scientific data. In addition, scientific data often requires complex calibration operations that must draw on large bodies of data about equipment and its performance in various states. The restrictions imposed by the relational model make it very difficult to deal with complex problems of this nature. Both these assumptions made by the relational model will be challenged by this new model.

The software interface to the database manager is assumed to be an object-oriented interface[3], implemented in C++[4]. The users of this software interface are C++ application

[1] Date, C., 1986, "Relational Database, Selected Writings", Addison-Wesley Publishing Company.

[2] "Implementation of the Flexible Image Transport System (FITS)", November 6, 1991, Draft Standard, NOST 100-0.3b, NASA/OSSA Office of Standards and Technology, Code 933, NASA Goddard Space Flight Center, Greenbelt, Maryland. Appendix A discusses the proposed binary tables extension to the FITS standard.

[3] Rumbaugh, G., 1991, "Object-Oriented Modeling and Design", Prentice Hall; Booch, G., 1991, "Object-Oriented Design", The Benjamin/Cummings Publishing Company, Inc.

[4] Stroustrup, B., 1991, "The C++ Programming Language", Second Edition, Addison-Wesley Publishing Company.

developers making use of the database management concepts to store and access data or C++ objects. While it is possible to conceive of other interfaces to the database manager, including stand-alone or end-user interfaces, they are not addressed here.

1.2 FITS Binary Tables

A FITS binary table, from an abstract perspective, is a collection of rows of data fields, organized into columns. This structure is familiar to spreadsheet and relational database users. The data fields under the same column all have similar attributes. A descriptive header is associated with the table. This header applies to and describes the table as a whole. There is also a set of keywords, each of which applies to the table as a whole. A keyword is a name plus a data value. The table header is a set of data items corresponding to a fixed set of attributes; all tables have this set of attributes. Keywords, on the other hand, are optional and apply only to a particular table. This structure is depicted in figure DBMANAGER0. There are several significant properties of such a table.

1. The table contains a fixed number of columns.
2. All rows have the same size, which is some fixed quantity.
3. The table may contain an unlimited number of rows.
4. The number of keywords is arbitrary and may be zero.

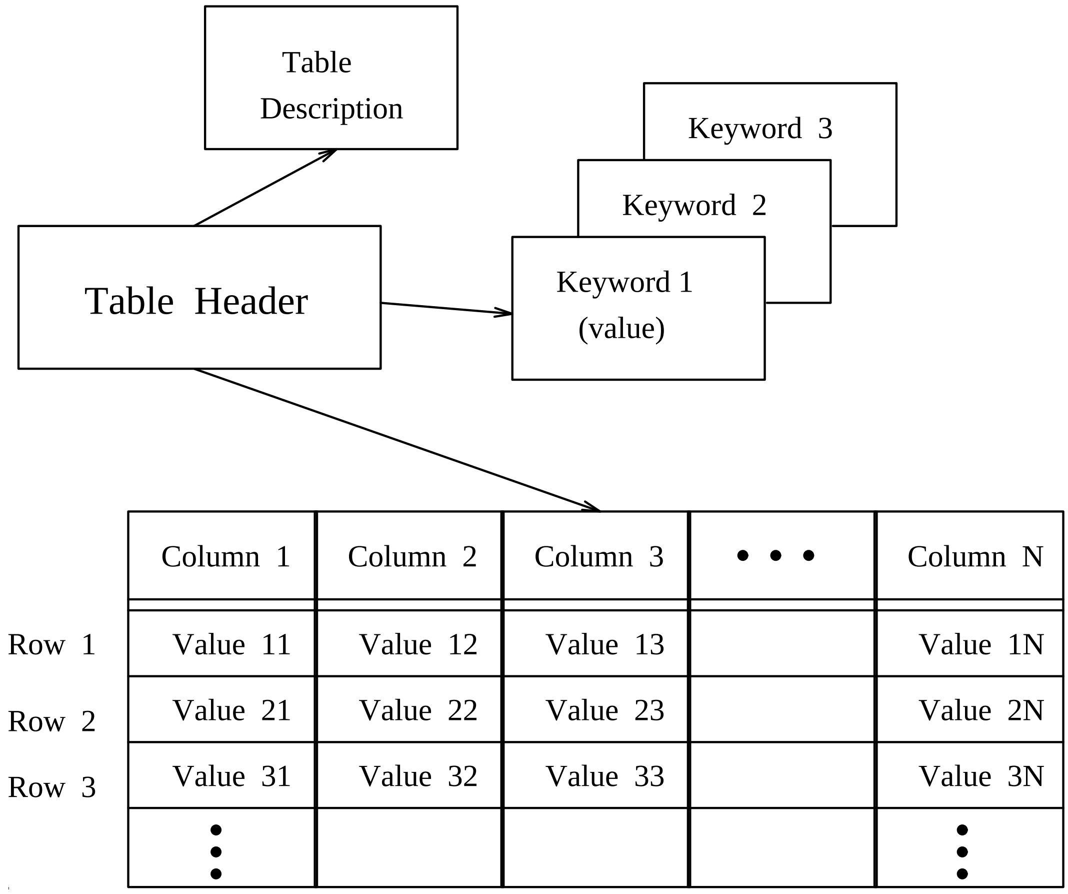

Figure DBMANAGER0 – FITS Binary Table.

We must extend and sharpen some of these basic concepts. We will begin by considering keywords. First, keywords are to be considered as fields, having the same properties as fields in a column of a table, *i.e.*, any data type or data property that can be predicated of a column can also be predicated of a keyword field. The only difference is that keywords can have only one field, while a column can have many rows. Both columns and keywords have names and a defined order within the definition of the table. Particular columns or keywords may be referenced either by name or by numerical order. Second, keywords may be attached to individual columns or keywords within a table, in addition to the table as a whole. There are cases in which one wishes to provide additional descriptive information about a table column or keyword. Such information can be placed into keyword fields attached to the relevant table column or table keyword. Keywords attached to the entire table should apply to the table as a whole and not to particular columns. Third, the concept of storage-mode will be added as a property of fields in the columns of a table, taking two values: 'direct' and 'indirect'. If the storage-mode is 'direct', which is assumed to be the case in most circumstances, the actual data item is stored in a row of the table. If the storage-mode is 'indirect', the database manager stores the actual data of the field in a separate location and stores a fixed-length pointer to the data in the row of the table. This mechanism allows one to store variable-length fields within a table, while retaining the concept of a table of fixed-length rows. Of course, an extra level of indirection is imposed on the variable-length fields. Under the assumption that most fields will have fixed sizes, this should not impose too great a performance penalty. This method of handling variable-length fields is consistent with the "Variable Length Array" facility described in section A.9.2 of the proposed FITS standard.

What are the data types of the fields to be supported? The database manager will support all the basic data types supported by FITS binary tables. These include: logical, unsigned, char, short, int, character strings, float, double, single-precision complex, and double-precision complex. The 'pointer' or 'variable length array descriptor' type, described in section A.6 of the proposed FITS standard, is being interpreted as a storage mode and not as a data type. FITS binary tables have an additional property that is important in this context and that will be supported. Any field may be a single element or may be a one-dimensional array of elements. Since keywords share the same properties as fields in a table, keywords also may be one-dimensional arrays of elements. At this point we have a data model that is reasonably close to standard FITS binary tables, with only a few conservative extensions to those basic concepts. These properties are summarized below. Figure DBMANAGER1 shows the possible structure of a table.

- A table consists of a header, an arbitrary number of keyword fields, and a fixed number of columns of fields organized into rows of a fixed size.
- Keyword fields have the same data types and properties as fields within columns.
- Keyword fields may be attached to individual columns or keywords or to the table as a whole.
- Fields within columns may be stored directly or indirectly.
- Fields may be single data items or one-dimensional arrays of data items of the following types: logical, char, unsigned char, short, unsigned short, int, unsigned int, character strings, float, double, single-precision complex, and double-precision complex.

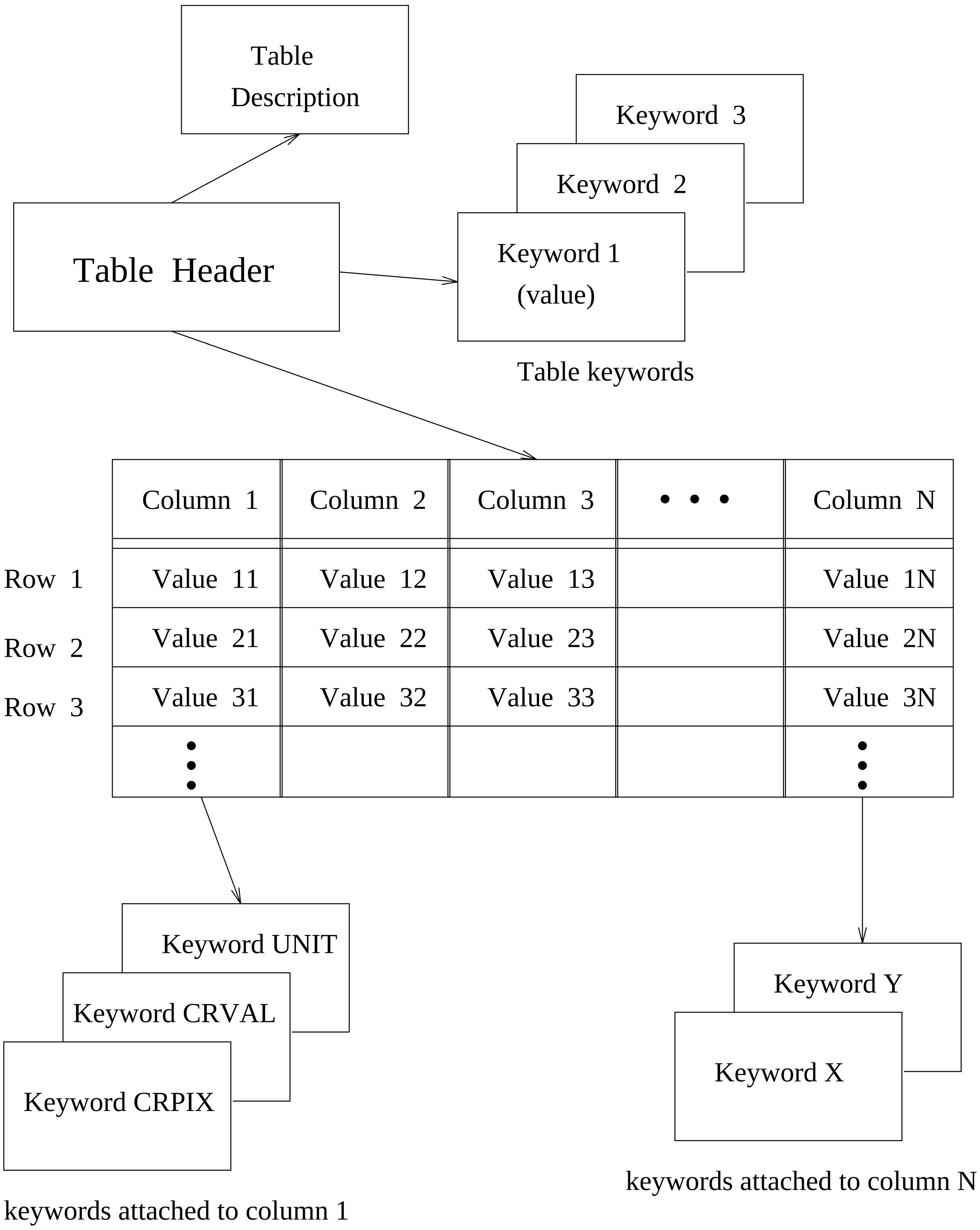

Figure DBMANAGER1 – AIPS++ Table.

1.3 Extensions to Table Concepts

There are a already a number of significant differences between tables, as presented here, and the tables of the relational database model. It is useful to pause and note what these are. First, relational database tables have no concept of keyword fields. Second, they have no concept of an indirect storage mode. Third, fields cannot be one-dimensional arrays. It is true that our list of supported data types includes some types not supported by most relational database management systems. However, the relational model does not specify a list of elementary data types; it merely specifies that they be atomic, etc. Even with the properties presented so far, these 'extended' tables are capable of modeling a broader range of relationships. In addition to being able to handle variable-length fields and one-dimensional arrays, keyword fields can be exploited to model one-to-many relationships. The list of keywords fields that attach to the table as a whole models the 'one' part of the relationship, while the rows of the table model the 'many' part of the relationship. The requirement that keyword fields support the same data types and properties as fields in columns is an essential property that makes such an association possible. If keywords are restricted in some fashion, then one cannot model all one-to-many relationships. We must now turn to further, not so conservative, extensions to our concept of a table.

We will consider four kinds of extensions. They are all related to the concept of a field and expand that concept. First, the data types of fields should be extensible. Second, fields can be multi-dimensional arrays. Third, fields can reference other tables. Fourth, columns within a table can reference previously defined row descriptions. We will now discuss each of these.

The data types of fields should be extensible, *i.e.*, the data model should be as independent of the list of supported data types as possible. In considering the database manager, this translates into being able to easily add new data types without performing major surgery on the software. Such additions may well be application specific and should be encouraged. In astronomy applications we may well want to add 'date', 'time', 'ra', and 'dec' as elementary data types. In commercial applications, we may want to add 'money' as an elementary data type. Such extensibility is adequately supported and encouraged by object-oriented programming and C++. Therefore, when we are considering a list of elementary data types, we should view this as a finite, but easily extensible, list of such types.
The type system in the tables will eventually also allow compound data types, *i.e.* C-like structs. Since a field can be an array of items, arrays of structures can be formed. The NRAO Green Bank spectral processor data[5] can be easily described in this manner.

The significance of multi-dimensional arrays to scientific data analysis has already been emphasized. They will be supported in the following manner. The description of the array, its structure, *i.e.*, the number of dimensions and the number of elements in each dimension, as well as other descriptive information will be stored separately by the database manager. Then, a field that is a multi-dimensional array specifies the array description. There is no limit to the number of dimensions supported. The 'storage mode' attribute applies to multi-dimensional arrays as well; 'direct' arrays are stored in the row of the table itself, while 'indirect' arrays are stored separately. A table containing multi-dimensional arrays is depicted in Figure DBMANAGER2. One must keep in mind the principle that keyword fields have the same properties as fields in columns. A consequence, of course, is that keyword fields may be multi-dimensional arrays in the same manner as fields in columns.

[5] M. Clark, 1990, "Green Bank Single-Dish Telescope Data Format", private communication, National Radio Astronomy Observatory, Green Bank, West Virginia.

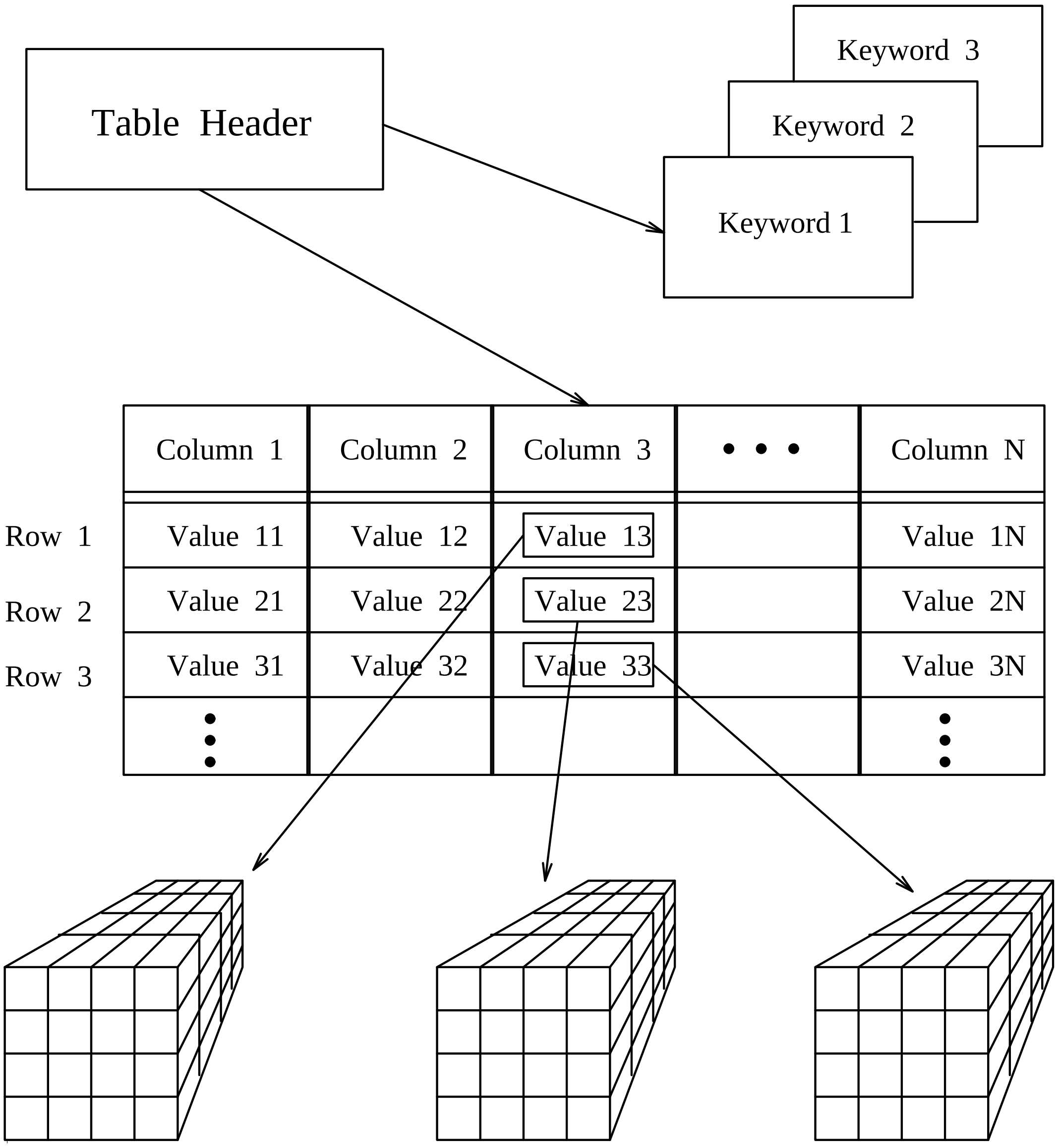

Figure DBMANAGER2 – Fields as Multi-Dimensional Arrays.

One of the obvious requirements of any data model is to be able to form relationships between various collections of data. In the relational model, all tables are independent of each other. Relationships between them are not a defined part of the tables themselves; rather, relationships are formed, at 'run time' by processing that relies on the contents of the tables involved. This is usually done through a query language of some sort, such as SQL[6]. One task that SQL accomplishes is to create new relational tables from data in existing tables. These are constructed by exploiting relationships between the existing tables that are expressed in the SQL statements. In other words, the relationships between tables are not part of the defined structure of the database tables; they are in the SQL statements. This mechanism has great flexibility and, in conjunction with other properties of the relational model, is the basis for its 'completeness', *i.e.*, that the query language, can, in principle,

[6] ANSI X3.135-1986, American National Standard for Information Systems,1986, "Database Language – SQL", American National Standards Institute, New York, New York.

form any new relationship that can be expressed in terms of first-order predicate logic and set theory. From a practical perspective, this way of handling relationships is successful when the relationships are relatively simple. However, when they become more complex, the SQL statements become complex to the point of incomprehensibility. Processing time is also a consideration; it matters little that the language is relationally complete if it requires an inordinate amount of time to process a query. Scientific data analysis demands efficient access to large bodies of complex data. It is precisely under these conditions that the relational model tends to break down.

We will confront the situation described above by allowing fields to reference other tables. This may sound like a relatively simple extension, but it has a dramatic affect on the kinds of relationships that can be formed. A table containing fields referencing other tables is depicted in Figure DBMANAGER3. Of course, the referenced tables may contain references to other tables, etc. In this manner, one can easily model natural hierarchies, e. g., tracking the installed peripherals and software packages on computer systems within divisions of a company. There is a one-to-many relationship between divisions of a company and computer systems; and, there are one-to-many relationships between computer systems and the installed peripherals, on the one hand, and the installed software packages, on the other hand. Keyword fields can also reference tables, as is depicted in Figure DBMANAGER4. We must also keep in mind that any field can be a one-dimensional array, enabling one to form arrays of tables. With a keyword field as an array of tables, one can form a hypertext-like linkage between a given table and many other tables.

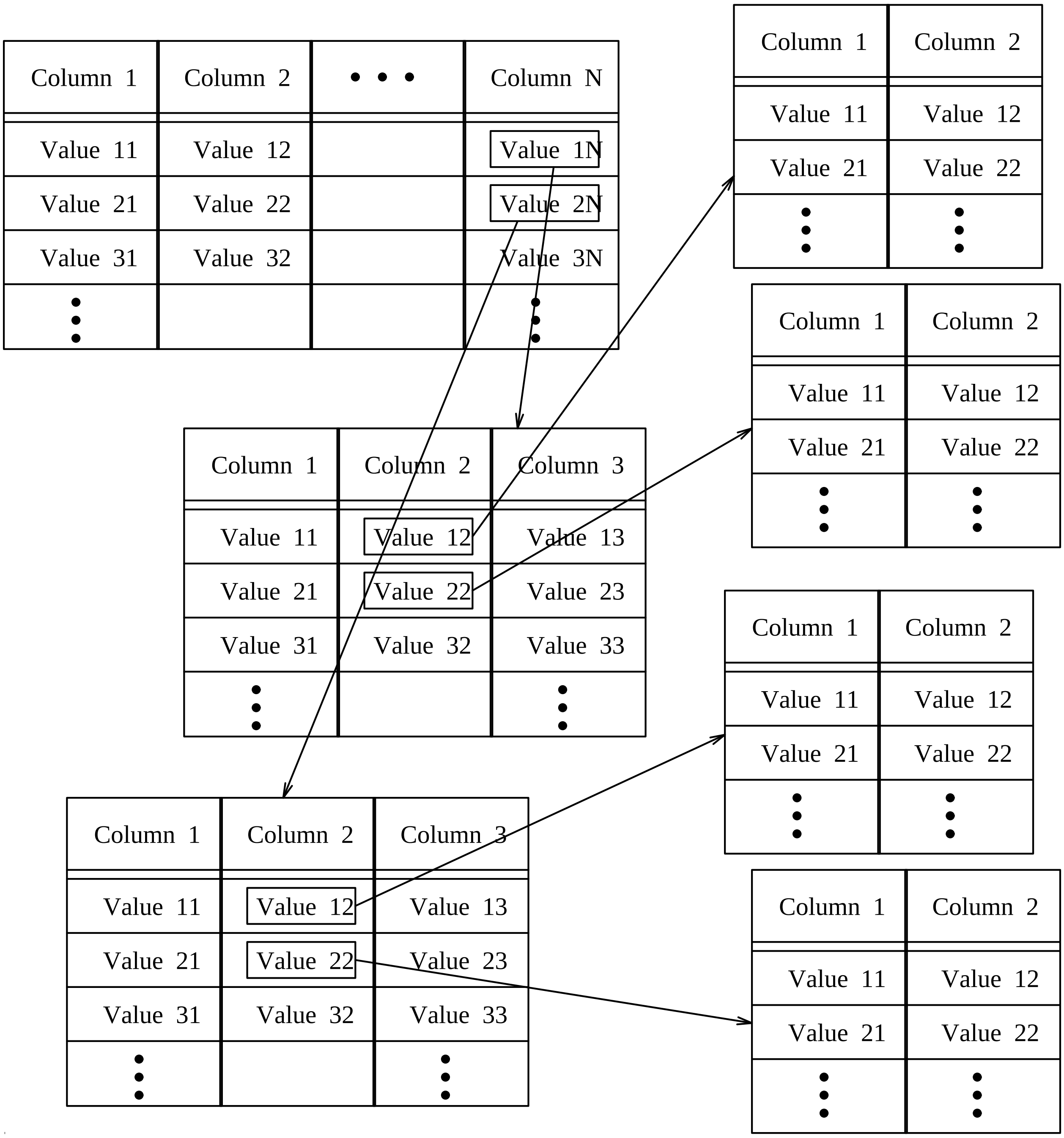

Figure DBMANAGER3 – Column Fields Referencing Tables.

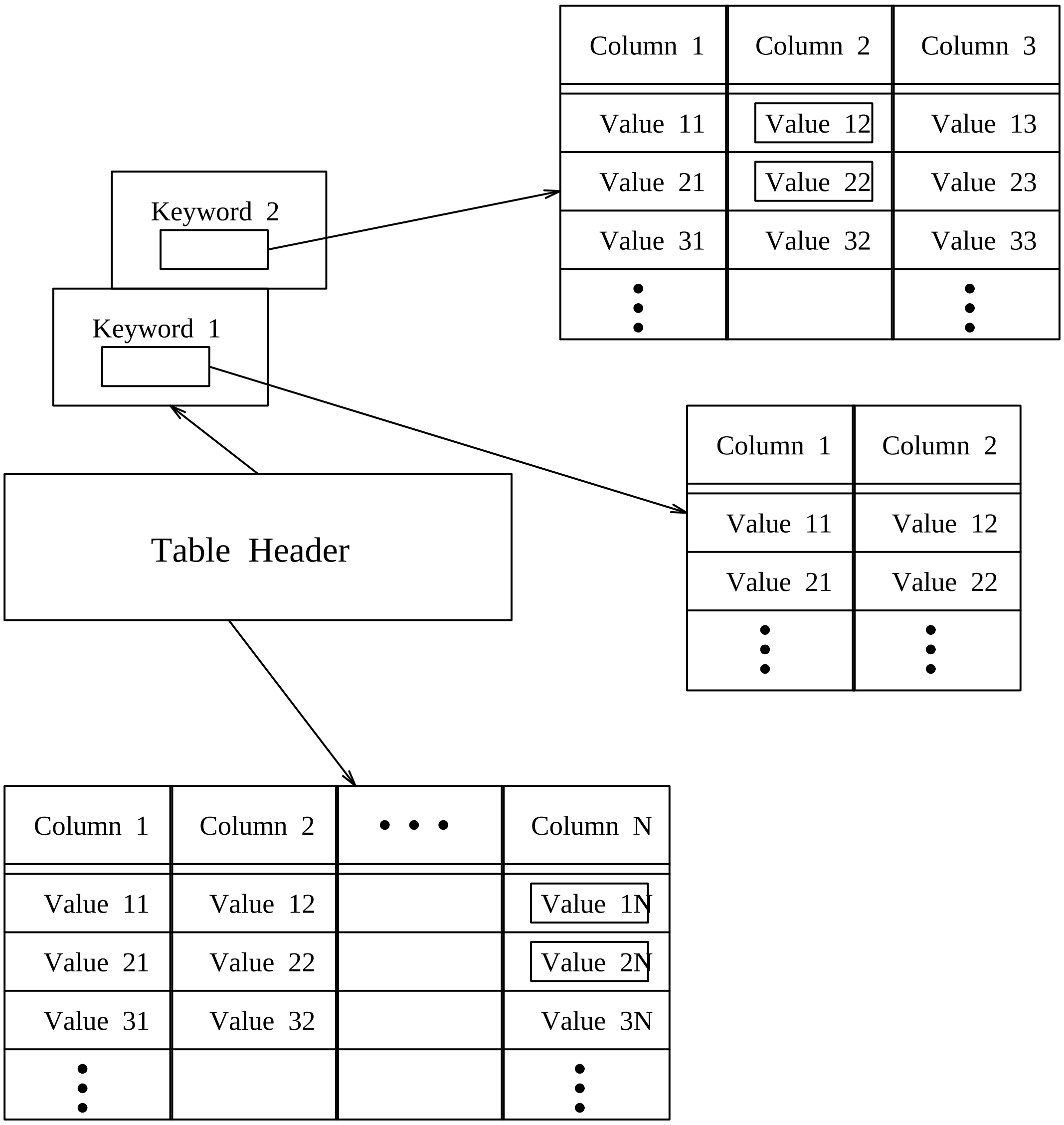

Figure DBMANAGER4 – Keyword Referencing Table.

1.4 VLBA Data as a Table of Tables

In this section we will use the proposed data format of the VLBA Correlator (Diamond and Wells 1992) to illustrate some of the concepts just discussed. The VLBA Correlator data format is presented as a sequence of FITS binary tables. A key component of this sequence is the UV table. Its rows contain data items familiar to interferometry, viz., U, V, W, date, time, baseline, array, source number, frequency id number, data integration time, weight, and finally the interferometer data. This interferometer data field is actually a 2x4x128x4 four-dimensional matrix, using the example from the paper. However, while this table is a key component, it requires several other tables for proper interpretation and processing, e. g., a flagging table, a calibration table, and an antenna table. Additional tables, which are constant over relatively long periods of time, are also required, e. g., an array geometry table, a frequency table, and a source table.

Using the concepts embodied in our 'extended' tables, the most natural approach to this VLBA data is to view it as a table of tables. This structure is presented in Figure DBMANAGER5. The VLBA data tables are separated into two groups, time-invariant tables and time-variant tables. The VLBA header points to a list of keyword fields that reference tables. These are assumed to be time-invariant and include the array geometry table, frequency table, source table, and gain curve table. The VLBA table itself is a sequence of columns, each of which references a time-variant table. The rows of this table, marked group 1, group 2, etc., are all the tables that are relevant to that period of time. These time-variant tables include the UV data table itself, antenna characteristics table, calibration table, flagging table, bandpass table, baseline correction table, phase-cal table, interferometer model table, CALC table, weather table, opacity table, ephemeris table, and sampler statistics table. The UV data table is the table of U, V, W, etc., indicated above. Its rows are ordered by increasing time.

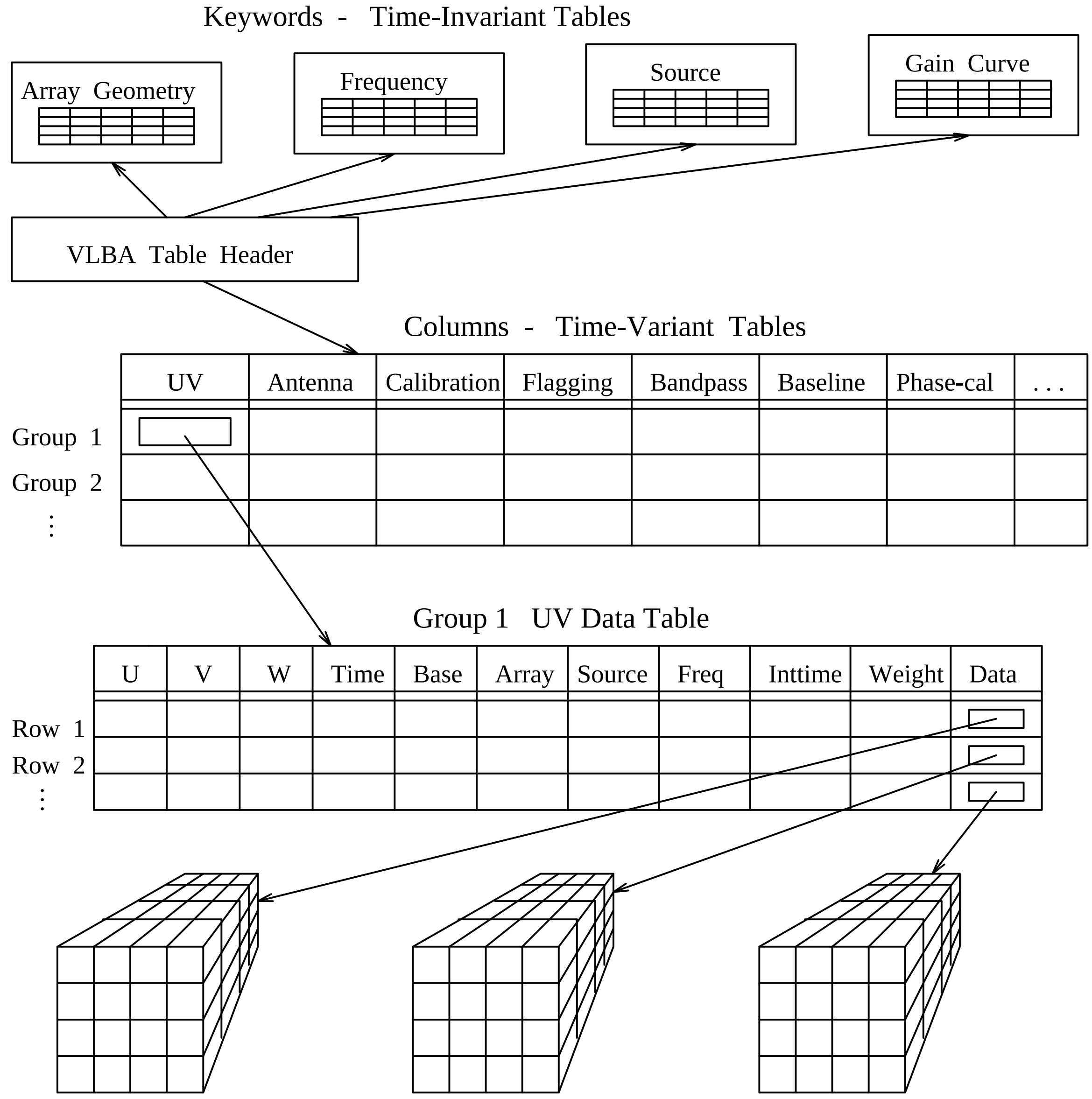

Figure DBMANAGER5 – VLBA Data as a Table of Tables.

It should be noted that the VLBA table in Figure DBMANAGER5 consists of a collection of references to other tables. As such, it is a high-level structure that expresses relationships between groups of tables. It is also fairly easy to change these relationships. Suppose that the gain curves were found to be unstable over long periods of time and the decision was made to move them from the time-invariant set of tables referenced by the keywords to a column referencing time-variant tables. It is a very easy task to restructure all existing VLBA tables stored in the database to conform to the new format. One does not have to restructure vast quantities of data; all that is required is to restructure a set of references to other tables, not the actual tables that are referenced.

1.5 Storage Concepts, Table Types, and Table Instantiations

Roughly speaking, the database manager is partitioned into the following subsystems: the data definition processor, storage manager, index manager, and data access/update manager. At this point we must discuss some of the requirements of the storage manager in order to introduce the concept of a table type. This will be followed by a description of the process of instantiating a table from a table type.

Whatever the storage manager is given to store, it considers to be an object. There are six kinds of objects that it must store and manage. These are: 1) table descriptions, 2) row descriptions, 3) multi-dimensional array descriptions, 4) table objects derived from table descriptions, 5) row objects that belong to table objects, and 6) fields within row objects that have indirect storage mode. Each stored object, regardless of the kind of object it is, is given a unique identifier, a UID. This UID is strictly internal to the storage manager. In order to properly manage the stored objects, the storage manager utilizes five directories: the UID directory, descriptor directory, table directory, row directory, and indirect field directory. Everything stored by the storage manager is entered into the UID directory. In addition, table, row and array descriptors are stored in the descriptor directory. Table objects derived from table descriptions are entered into the table directory. Rows belonging to table objects are entered into the row directory, and indirect fields are entered into the indirect field directory.

There is much more to be specified about the storage manager. A great deal of thought must be given to the problem of storage optimization schemes. This becomes particularly accute, since scientific data analysis tends to require rapid manipulation and modification of large volumes of data. The storage manager must be sufficiently flexible to accomodate different types of storage mechanisms. However, such flexibility and optimization schemes should remain encapsulated within the storage manager. The storage manager should also utilize the C++ technique of 'smart pointers' to resolve references to tables or indirect fields. Not only does this technique simplify the interface to the storage manager, it serves to insulate access methods that should remain confined to the storage manager itself. Much more could be said, but this will suffice for our purposes.

The database manager contains all the information necessary to understand the structure of everything stored in the database, regardless of its complexity. A consequence of this fact is that the database manager is capable of generating C++ data declaration statements for use in application programs. An example of such output is given beloe, which is a declaration of the UV data matrix from the VLBA example discussed in section 4. Every table is accompanied by a descriptor that contains all descriptive information about that table; a complete specification of the properties of its columns and its keywords. This is an essential aspect of the database manager.

Within this scheme there is a natural way to form the concept of classes of tables. We must recall the concept of a field. A field, in general, is an array of elements. The number of elements in the field need not be specified until we actually construct a table. If the number of elements in the field is left unspecified in the table descriptor, then it may take any value when a table is actually constructed using the descriptor. This is true for column fields and for keyword fields. Keyword fields have a data value associated with them. The elementary type of this data item must be specified but its actual value need not be specified in the table descriptor. If such a value is specified, it may be overridden when an actual table is constructed using the table descriptor. In this manner, a keyword data value in a keyword of a table descriptor becomes a default value for any table created from that descriptor. Such table descriptors become descriptions of classes of tables, viz., all those tables that have been created using that particular table description. This gives us a mechanism for defining such a table descriptor as a table type. A particular table created from that table type is said to be an **instantiation** of that table type. This use of the term 'instantiation' is in strict conformance to its meaning in object-oriented programming. There are several advantages to such a concept. It can be used to refine and restrict search operations; one can search tables of a particular type rather than all tables. It can also be used to simplfy and generalize application programs. Through a suitable scheme of C++ operator overloading (Stroustrup 1991, pp 225-254), operators can be defined in such a manner that application programs are not dependent on the sizes of arrays. This means that application programs need not be dependent on a fixed number of frequency channels or time samples, a problem that has plagued data analysis programs in the past. Finally, this concept of a table type can be used as an integrity check. If a field references a table, then, in the table descriptor, that field specifies a table type. When an instantiated table assigns a table reference to that field, it must conform to that table type. This mechanism can be used to preserve data integrity, in the same manner as specifying an elementary data type.

One additional remark must be made about the storage manager. Preserving data integrity is an important consideration in a database. Allowing fields to reference other tables creates a problem in this respect. If a referenced table is deleted, one is left with a 'dangling' reference, one that no longer references a valid object. In order to deal with this problem, the storage manager must maintain a 'where-referenced' directory and either refuse to delete a table that is referenced or redirect the dangling reference to some appropriate error-handling mechanism.

1.6 Database Operations

We will now give a brief overview of some of the database operations from a user's perspective. These may be grouped into five categories: 1) manipulating descriptions, 2) instantiating and destroying tables, 3) modifying the contents of a table, 4) indexing tables, and 5) searching for tables. With respect to manipulating descriptions, one must be able to store and reference a table description, a row description and an array description. Likewise, one must be able to remove these. The process of instantiating and destroying tables, *i.e.*, creating a table object from a table description and removing an entire table, is special because a lot of internal checking and bookkeeping takes place. In order to modify the contents of a table, one must be able to access and/or modify the rows of a table object, add a row to a table object, and delete a row from a table. The ability to index tables is especially significant. The database manager must have considerable flexibility in this respect. One must be able to form indices to tables based on header values and keyword values. One must also be able to form indices to rows of tables based on column values. There must also be different types of indexing schemes. Classic B+-tree or extendible hashing indices will be needed, but so will others. Within astronomy, longitude and latitude data will require some sort of spatial indexing scheme. Finally, one must be able to utilize these indexing mechanisms to search for tables or for rows of a table or tables. One should have the ability to direct that search to tables of a specified type.

1.7 Additional Work

There are a number of issues that have not been addressed and remain the subject of future investigation. First, the subject of storage optimization schemes has already been mentioned. Second, the specific form of the search capability needs to be investigated, as well as its user interface. This search capability should be able to exploit the concept of a table type in useful ways, in order to cope with higher-level relationships and structures within the database. Third, considerable attention needs to be given to creating new table objects from subsets and combinations of existing table objects. This is particularly important in a scientific data analysis context. Of course, one must be able to select columns and rows of a table, operations that are familiar in a database context. Additionally, one must be able to select subsets or various types of projections of multi-dimensional arrays. The challenge will be to find a convenient user interface and efficient mechanism for accomplishing such operations.

1.8 Conclusions

To summarize, a complete list of the properties of tables, as they have been presented, is provided below.

- A table consists of a header, an arbitrary number of keyword fields, and a fixed number of columns of fields organized into rows of a fixed size.
- Keyword fields have the same set of data types and properties as fields within columns.
- Keyword fields may be attached to individual columns or keywords or to the table as a whole.
- Fields within columns may be stored directly or indirectly.
- Fields may be single data items or multi-dimensional arrays of data items.
- The data types of the fields are an extensible set of types, but include the following: logical, unsigned, char, short, int, character strings, float, double, single-precision complex, and double-precision complex.
- Fields can reference other tables.

2 Table Overview

The AIPS++ table data system closely follows the table model described in the first chapter. Thus a table consists of a set of keywords and a set of columns. Each keyword and column value can be a scalar, an array or a table. All basic data types, including string and complex, are supported. Arrays and tables can be direct or indirect values. Direct arrays and tables are directly contained in the table and must have the same shape in all rows. Indirect arrays and tables are only referred to and they can have different shapes in different rows. A set of keywords attached to a table column or keyword can, for example, keep information such as scale-factor, unit, crpix and crval.

2.1 Table Types

The AIPS++ table data system is implemented as a generic system which can include different types of tables. This is done by defining abstract base classes, from which other classes are derived. However, except for opening a table, the programmer will not see any difference because access to all tables is performed the same way. The possible table types are:

1. Filled Tables
   These tables contain real data which can be held in files. Each filled table needs a table description defining the structure of the table before the filled table can be created.

   A storage manager stores the data of a filled table into a file. Different storage managers can exist, possibly specialized for a particular table kind (*e.g.* UV-data, star catalogue). Currently only a simple storage manager using the AipsIO class exists, but in the near future more elaborate storage managers will be developed.
2. Reference Tables
   These tables reference rows in other tables. They are the result of a select, sort or iteration operation on a table. They can be stored in a file to keep them for later usage.
3. Virtual Tables
   These tables contain no real data, but their data can be assembled at run-time from other tables or generated on-the-fly. It is partly similar to the concept of a view in a relational data base system. It can, for instance, be used to view calibrated data as a table, which will be composed at run-time from uncalibrated data and calibration tables. It can also be used to implement an artificial observation as a table by calculating the data on-the-fly.

   There can be many types of virtual tables, because each one can be implemented in its own way.

2.2 Main Classes

A few classes form the foundation of the table system. Figure TABCLASS1 contains a Rumbaugh diagram showing their relations.

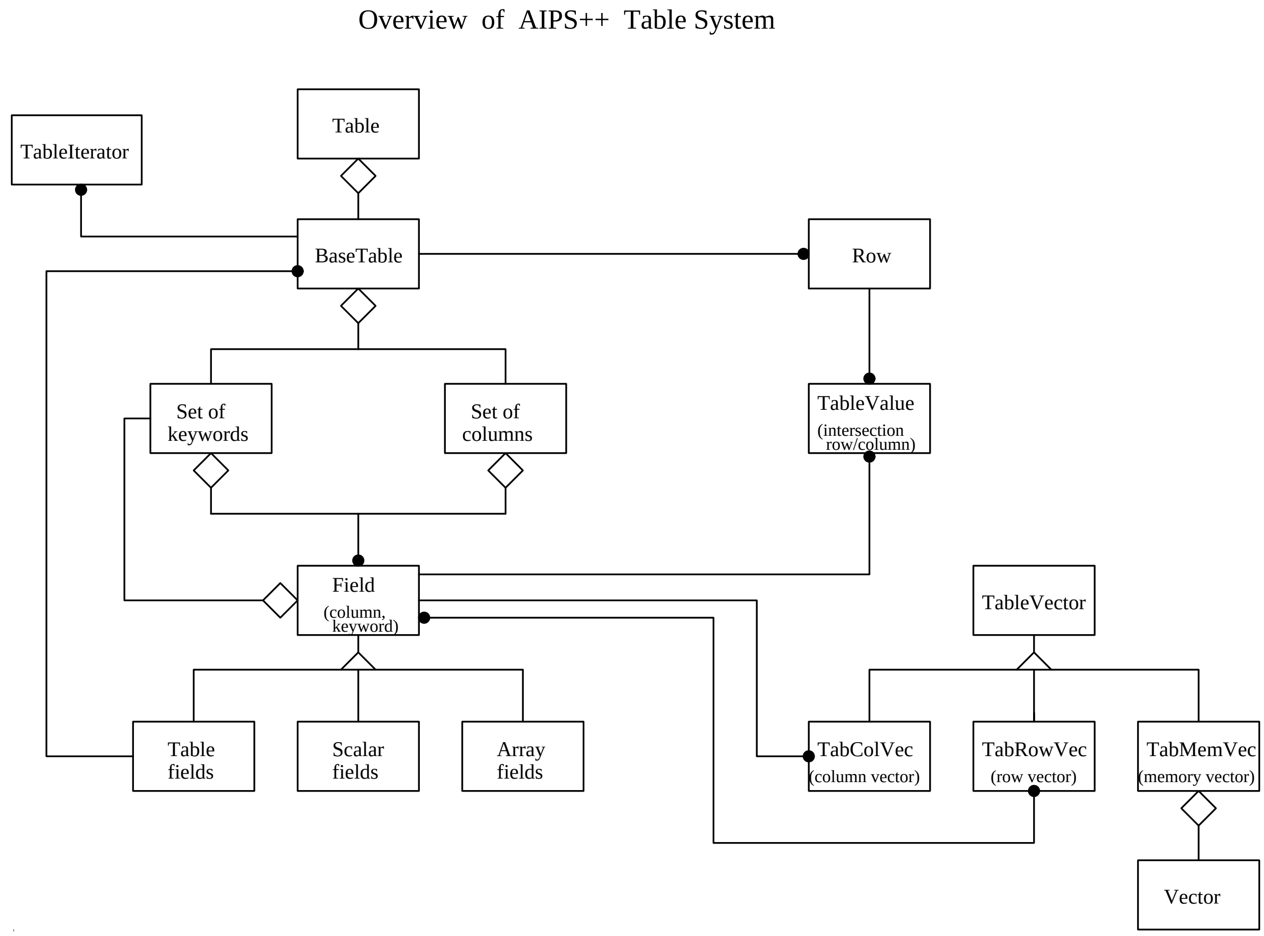

Figure TABCLASS1 – Table Base Classes.

This diagram shows the general structure of the table system. It does not show the classes TabDesc and FieldDesc. They are needed to define a description for a filled table, thus are special for filled tables. However, since filled tables are an important kind of tables, they are mentioned in the following summary of the main table classes.

`TabDesc, FieldDesc`
: These classes are needed to define a description of a table and the keywords and columns in it.

`Table`
: This class represents a table. An individual keyword or column is represented by class `Field` (see below). `Table` has to be used to get access to the set of table keywords or to a row in the table columns. `Table` also contains functions to construct another table by doing a select, sort or iteration.

`BaseTable`
: This class will not be used by most users. It is only needed when a virtual table class is implemented. It is discussed in more detail in ⟨undefined⟩ [(], page ⟨undefined⟩Detailed Implementation).

`Field`
: This class represents a keyword or a column in a table. It has to be used to get access to the set of keywords attached to a table column or keyword. Furthermore it has to be used to get access to an entire column.

`Row` This class represents a row in the table columns. It also represents a set of keyword values (be it table keywords or keywords attached to a table column or keyword). It has to be used to get access to a table value.

`TableValue`
: This class represents a table value (be it in a column or keyword). It can be used to get or put a scalar or array(-slice) from/into a table value. It can also be used to get access to a nested table (*i.e.* if a field contains a table).

`TableVector`
: This templated class allows to handle a table column or row as a vector. All operations defined on the usual `Vector` class (*e.g.* `>`, *, `+=`, `sin`, `dotproduct`) are also defined for `TableVector`.

`TableIterator`
: This class allows to iterate through a table in an arbitrary way. Multiple iterators can exist for the same table. For each iteration step a subtable will be returned in which the values in the iteration columns are the same.

The following chapters describe in more detail how these classes can be used to manipulate data in tables. In short, the following operations are possible:

- Creating, reading and updating a table description.
- Creating, reading and updating a table.
- Getting and putting data from/into a single value.
- Getting and putting slices of an array contained in a value.
- Getting and putting data from/into an entire column.
- Handling a column as a vector.
- Adding, removing and renaming keywords and columns.
- Selecting rows from a table.
- Sorting a table on one or more columns.
- Iterating through a table in an arbitrary order.

Appendix A [Examples], page 48 contains some examples how to use the table classes and functions

Often the envelope/letter idiom[1] is used in the table classes, but the regular user will not notice that. Only advanced users implementing virtual tables must be aware of the underlying letter classes and derive from them in a correct way. See Chapter 9 [Detailed Implementation], page 41 for more information on how to implement classes derived from the base table classes.

2.3 Error Handling

When a table function encounters an error, it will throw an exception. Exceptions can be caught by a `try-catch` block as described in the document on the AIPS++ exception handling.

[1] Coplien, James O., 1992, "Advanced C++ Programming Styles and Idioms", pp. 133-164, Addison-Wesley Publishing Company

Table specific exceptions are defined in the header file `TableError.h`. Apart from these exceptions, exceptions from other classes (*e.g.* `Array`, `OrderedMap`) can also be thrown.

2.4 Header Files

Several header files have to be included to be able to use the tables classes and objects.

`TabDesc.h`
`FieldDesc.h`
: are needed to create a table description.

`Table.h`
`Row.h`
`TableValue.h`
`Field.h`
: are needed for all basic table operations.

`TabExprNode.h`
: is needed to select rows from a table.

`TableIter.h`
: is needed to create and use a table iterator.

`TableVector.h`
: is needed to construct a `TableVector` and to do basic operations on it.

`TabVecMath.h`
: is needed to do mathematical operations on a table vector.

`TabVecLogic.h`
: is needed to do relational operations (`>`, ==, etc.) on a table vector.

`TableError.h`
: is needed to catch an exception thrown by a function in one of the table classes.

Of course, header files like `Vector.h` and `ArrayMath.h` will also be needed if vectors, mathematical operations on arrays, etc. are used.

3 Table Description

3.1 Open/Create

A table description has to be defined before a filled table can be created. The description defines all keywords and columns in the table and attributes associated with them. A virtual table may also need a description (this is up to the designer of that virtual table class). When a table is created, it makes a copy of the description. This means that the original description can be changed without affecting tables created from it.

The class `TabDesc` is the basic class for defining a table description. It is possible to keep this description in a file to be reused. A description must have a name which can be of any length, although a scratch description can have an empty name (`""`).
The `TabDesc` constructor creates a description.

```
TabDesc td ("XYZ", TabDesc::New);
```

sets up a new (empty) description for the table type `XYZ`.

The available `TabDesc` options are:

`Old` read an description (this is the default); exception if it does not exist

`Update` update an existing description; exception if it does not exist

`New` create a new description; overwrite if it exist;

`NewNoReplace`
create a new description; exception if it exists

`Scratch` create a temporary description; exception if it exists

`Delete` delete an existing description; exception if it does not exist

A non-scratch table description will be held in the file `X.tabdesc`, where `X` is the name of the table description. A new file will be located in the default directory, while for an existing one a search path can be used.
The `TabDesc` destructor writes a new or updated description into the file using `AipsIO`. Also the destructor deletes a description to be deleted. If the destructor is executed due to an exception, it will not write nor delete. Note that a modified description opened with option `Old`, will **not** be written.

There are 2 ways in which a table can be defined, which can be combined.

1. In a static way by first filling a table description (*i.e.* adding keywords and/or columns to it) and then creating the table from the description. This description is generic in the sense that is not necessary yet to define the shape of arrays. The next sections describe how a table description can be filled and updated.
2. In a dynamic way by creating a table from an description (which can be empty) and adding keywords and/or columns to the table. In this way the description is augmented on-the-fly. This will, for instance, be needed when a FITS file is read. This also allows for renaming and removing of keywords and columns.

3.2 Defining Keyword and Columns

Keywords and columns can be added to a table description using the `TabDesc` functions `addKey` and `addCol`, respectivily. These functions have an often optional argument `options` in which options for that field can be specified. There are several options, which can be or-ed.

`FieldDesc::Direct`
: This specifies that a keyword or column containing an array or table has to be direct, thus stored in the table itself. If this option is not given it will be indirect, thus the table only contains a reference to the array or table. See Section 5.3 [Shape/Origin], page 27, for more information.

`FieldDesc::VarDesc`
: This specifies that the tables in a column can have different descriptions per row, thus keywords or columns can be added to a table in a specific row. If this option is not given, the tables in all rows of a column must have the same description. Direct tables must have the same description, so `FieldDesc::Direct` overrides `FieldDesc::VarDesc`. See Chapter 6 [Changing Tables], page 31, for more information.

`FieldDesc::RootIndexed`
: This specifies that an index must be maintained for this field. However, it is not implemented yet.

`FieldDesc::Default`
: Use the default options, which are indirect, no varying descriptions and no index. If this is or-ed with another option, the other option will prevail.

The instances of the `addKey` (and `addCol`) functions are:

`addKey (const String& name, const DataType dt);`
: adds a keyword containing a scalar value, for example:

```
td.addKey ("RA", TpDouble);
```

adds the scalar keyword `RA` with data type double.

`addKey (const String& name, const uInt ndim, const DataType dt, const int option);`
: adds a keyword containing an array of values, for example:

```
td.addKey ("arr", 2, TpInt);
```

adds keyword `arr` being a 2-dimensional Int array.
`Ndim=0` means that the dimensionality of the arrays is not yet known. Indirect arrays can then have a different dimensionality in different rows. The optional 4th argument describes the options. Specify `FieldDesc::Direct` if the array has to be direct.
Other instances of the `addKey` and `addCol` functions for defining array keywords and columns allow for (part of) the shape and/or origin of the array to be defined (see classes Array and Vector for a definition of the shape and origin IPosition vectors). These are (in abbreviated form):

`addKey (name, ndim, dt, option, const IPosition& shape);`
: Uses zero origin.

`addKey (name, ndim, dt, option, shape, const IPosition& origin);`

`addKey (name, ndim, dt, option, const uInt x_size)`
Shorthand for a 1-dimensional array; uses zero origin.

`addKey (name, ndim, dt, option, x_size, y_size)`
Shorthand for a 2-dimensional array; uses zero origin.

`addKey (name, ndim, dt, option, x_size, y_size, z_size)`
Shorthand for a 3-dimensional array; uses zero origin.

Setting a dimension size in the shape vector to zero means it is undefined; trailing undefined dimensions can be left out, so the length of the shape vector can be less than the dimensionality of the array. The undefined dimension sizes must be set at run-time. The origin defaults to 0. If the origin is given, its vector must have the same length as the shape. If a dimension size in the shape is undefined, the corresponding origin is also undefined.

`addKey (const String&, const String&, const int, const uInt nrow = 0);`
adds a table keyword, for example:

```
td.addKey ("tab", "ABC");
```

adds keyword `tab` being a table of type `ABC`. The table type does not need to exist yet; only when a table is created from the description, the system looks for a file containing table description `ABC`.
If the table has a fixed length, the number of rows can be specified in the optional 4th argument. The default 0 means variable number of rows.

`addKey (const String&, const TabDesc&, const int, const uInt nrow = 0);`
is a second way to add a table keyword.

```
TabDesc tdsub("samplesub");
td.addKey ("tab",tdsub);
```

This adds keyword `tab` being a table of type `samplesub`. In this way the given table description is copied, which means that later changes made to `samplesub` will not be reflected into the table description keyword `tab` is pointing to.

`addKey (const FieldDesc*, const String& name);`
This clones the field defined by the field descriptor and adds it as a keyword with the given name to the table. Also all the attributes and possible keywords attached to the field are cloned. See Section 3.3 [Attributes], page 22, for defining attributes and attaching keywords.

3.3 Keyword and Column Attributes

The functions `addKey` and `addCol` return a pointer to the field description. This can be used to define additional attributes for that keyword or column. The following can optionally be defined:

‘`comment`’ A simple explanation of the keyword or column. This will be shown in a listing of the table. It can also be used by a FITS writer.

'`default value(s)`'

Default values can be defined for a scalar and for part or all of an array. The data type of the default should match the field data type. Defining default values for a part of an array means that the same defaults are used for each slice, which will be a pixel, row, *etc.*, depending on the number defined. Thus `1` (every pixel), `3` (every row), `3*4` (every plane) or `3*4*5` defaults can be defined for an array with shape `(3,4,5)`.

'`keywords`'

These keywords can be used to hold special information about the keyword or column they belong to. They can hold a unit, a scale factor, the FITS-value crpix, *etc.*. These keywords are treated in exactly the same manner as table keywords and columns, so comment, default(s) and keywords can also be defined for them. These keywords can be used for special purposes. A layer above the table system could interpret their values and, for example, scale a data array.

```
FieldDesc* fdp = td.addcol ("arr", 2, TpInt);
fdp->setComment ("Data array");
fdp->addKey ("crpix", TpFloat);
FieldDesc* fdp2 = fdp->addkey ("scale", TpFloat);
fdp2->setComment ("Scale-factor for the array");
fdp2->setDef ((float)1.0);
```

In the above example keywords `crpix` and `scale` are added. For the keyword `scale` the comment and default are defined. Note that the signature of this `FieldDesc` function `addKey` is the same as the `TabDesc` function `addKey`.

3.4 Renaming/Removing Keywords and Columns

After being defined, keywords can be renamed or removed using the `TabDesc` functions `renameKey` and `removeKey`, respectivily.

```
td.renameKey ("new", "old");      // rename keyword old to new
td.removeKey ("key");             // remove keyword key
```

Similarly the `TabDesc` functions `renameCol` and `removeCol` rename and remove a column.
The `FieldDesc` functions `renameKey` and `removeKey` rename and remove a keyword attached to a keyword or column.

3.5 Listing a Table Description.

A simple listing of the description can be made on `stdout` using the `TabDesc` function `show`.

```
td.show();          // list table description in td
```

4 Filled Tables

4.1 Introduction

A filled table is a table containing real data, which are stored in one or more files. A table description, described in the previous section, is needed to create a filled table. Since a filled table keeps its own copy of its description, it is possible to add, remove or rename keywords and columns in a dynamic way.

The current IO model for tables is very simple; tables will be held in memory and be read and written as a whole using the AipsIO class. When the prototype is finished an elaborate IO scheme will be developed.

4.2 Creating/Opening a Filled Table

As mentioned before, the class `Table` is to be used for all table handling including filled tables. Constructors of the class `Table` can create a new filled table and open an existing one. A filled table will be written or deleted by its destructor.

```
Table tab ("sample", "XYZ", Table::New);
```

creates a new filled table with file name `sample` using description `XYZ`.

```
Table tab ("subdir/sample2");
```

reads in the existing table file `subdir/sample2`.

The available `Table` options are:

`Old` read an existing table (this is the default); exception if it does not exist

`Update` update an existing table; exception if it does not exist

`New` create a new table; overwrite if it exists

`NewNoReplace` create a new table; exception if it exists

`Scratch` create a temporary table; exception if it exists

`Delete` delete an existing table; exception if it does not exist

5 Reading/Writing a Table

5.1 Accessing a Table

The basic classes for accessing data in a table are:

`Row` A set of all column values in a particular row. The set of all keywords in a table is also a `Row`. It is meant to get access to the `TableValue` objects. A `Row` object can be created by operations on the `Table` object.

```
Row row = tab.keywords();       // row of keywords in the table
Row row = tab[i];               // row i in the table
Row row = tab.addRow();         // add a row to the table
```

By default the `addRow` function adds one row to the table, which will be initialized (*i.e.* filled with default values). It is, however, possible to add more than row at a time and to skip initializing. If adding more than one row, the `Row` object returned is the first row added.

`TableValue`
A keyword or a particular value (*i.e.* row) in a column. It can be used to put or get a value or to get access to a nested table. A `TableValue` object can be constructed by indexing the row with the name of the keyword or column.

```
TableValue f = row["RA"];       // get access to RA this row
TableValue farr = row["arr"];   // get access to arr in this row
```

`Field` The set of all values in a keyword or column. It should be used to get or put an entire column. It should also be used to get access to the keywords belonging to a keyword or column. A pointer to `Field` can be obtained using the functions `getKey` and `getCol` in the `Table` object. Also the `[]` operator can be used to get either a keyword or column. Since a keyword and a column cannot have the same name, the result of `[]` is unambiguous.

```
Field* cfp = tab.getCol("RA");  // pointer to Field of col. RA
Field* cfp = tab["DEC"];        // pointer to Field of DEC
```

For columns, it would be inefficient to index a field with its name in each row. Therefore a field index number can be obtained with the `Table` function `getColIndex`:

```
Int inxra = tab.getColIndex ("RA");
```

and it can thereafter be used in:

```
TableValue f = row[inxra];      // get access to RA in this row
```

An index can also be used to get a pointer to a `Field` object.

```
Field* cfp = tab.getCol (inxra);
```

Similarly, the index number of a keyword can be obtained with:

```
Int inxkey = tab.getKeyIndex (name);
```

The following example shows how a loop through all rows or through all columns in a table can be written.

```
Table tab("SAMPLE");                    // open table SAMPLE for input
Row row;
Int inxra  = tab.getColIndex ("RA");
Int inxdec = tab.getColIndex ("DEC");
double ra, dec;
for (uInt i=0; i<tab.nrow(); i++) {   // loop through all rows
    row = tab[i];                       // get row i
    row[inxra].get (ra);                // get RA in this row
    row[inxdec].get (dec);              // get DEC in this row
    cout << ra << " " << dec << endl;
}

for (Int j=0; j<tab.ncol(); j++) {    // loop through all columns
    cout << tab.getCol(j)->getName() << endl; // show column name
}
```

5.2 Getting and Putting Values

Puts and gets can be done to/from a value in a row.

```
TableValue f = row["RA"];
TableValue farr = row["arr"];
f.put (3.14);                       // put a scalar value
Matrix<Int> arr(3,4);
farr.get (arr);                     // get a 2-dim array
```

Getting or putting a slice of an array can be done with the functions `getSlice` and `putSlice` in class `TableValue`. These functions come with several signatures to support the *blc, trc, inc* notation for general arrays and the _ notation (*start, end, increment*) for vectors, matrices and cubes.
Note that according to the `Array` class, a slice has the same dimensionality as the array, even if, say, only a row is taken out of a cube. The `Array` function `nonDegenerate` can be used to lower the dimensionality.

```
Array<double> arrd;
farrd.getSlice (arrd, _(10,100,2), _(i));    // get a slice
Vector<double> vecd = arrd.nonDegenerate();  // use as vector
```

The shape of an array or slice being put must conform the shape of the table array(-slice) (see class `Array` for conformation rules). However, when an array or slice is gotten, the receiving array can also be empty (*i.e.* have length zero) causing it to be resized.

The dimensionality, shape and origin of a table array can be obtained with:

```
uInt nd = f.ndim();
IPosition sh, or;
sh = f.shape();
or = f.origin();
```

If necessary, data type promotion will be done when a scalar value is accessed. So an Int keyword can, for example, be gotten as a double. The data type of an array has to match exactly. Before an array can be put, the dimensions of the array in the value have to be known. This is discussed in the next section.

Note that usually the `TableValue` object will be used only implicitly by combining the `Row` and `TableValue` functions as in:

```
row["RA"].put (3.14);
```

5.3 Setting/Getting Shape and Origin

The shape (and origin) of an array can be defined fully or partly when creating the description of the table. If it is not fully defined, the information need to be complemented before an array can be put. The array type (direct or indirect) determines when the shape and origin need to be defined.

- Direct arrays in a column must have the same shape in all rows. This means the full shape and origin has to be defined for the arrays in the entire column before any row is added to the table.
- Indirect arrays in a column can have different shapes and origins (and even different dimensionalities) in different rows. This means the shape and origin can be defined per row. However, they can also be defined fully or partly before any row is added to the table.

When direct tables are used in a column, their shape (*i.e.* number of rows) also has to be defined before any row is added to the table. However, the shape of indirect tables does not need to be set.

5.3.1 Direct Arrays

The shape and origin of the direct arrays (and tables) in a field has to be fully defined before the first row is added to the table. The `Field` function `setDim` has to be used for this purpose. This function comes in 5 versions.

`setDim (const IPosition&, const IPosition&);`
: to define shape and origin.

`setDim (const IPosition&);`
: to define the shape and use default origin 0.

`setDim (const uInt);`
`setDim (const uInt, const uInt);`
`setDim (const uInt, const uInt, const uInt);`
: to define the shape of a vector, matrix and cube, respectivily and to use default origin 0.

```
Field* cfp = tab.getCol("arr");
cfp->setDim (512,512);        // set shape for all arrays in column
```

or better combined in one statement:

```
tab.getCol("arr")->setDim (512,512);
```

If the size and origin of a dimension was already defined when defining the table description, they do not need to be defined again (use a zero for that dimension in the shape vector). If defined, they should match the predefined values in the description.

The `Field` functions `ndim`, `shape` and `origin` allow the "global" dimensionality, shape and origin of the values in a column to be obtained. Note that dimensionality is also defined for a scalar (=0).

5.3.2 Indirect Arrays

The shape and origin of indirect arrays can be different per row, which means they can be defined per row. However, some or all dimensions of all arrays in the column can be defined globally using the `TableValue` function `setDim` as described in the previous section on Direct Arrays. Note that if a zero dimensionality is defined in the description, the dimensionality can be different in each row.

To set the shape and origin per value (*i.e.* per row), the `TableValue` function `setDim` needs to be used. This function comes in 5 flavours with exactly the same signature as the corresponding `Field` functions described in the previous section.

```
row["arr"].setDim (1024,1024);  // set shape for value in this row
```

The `TableValue` functions `ndim`, `shape` and `origin` get the dimensionality, shape and origin of a value in a particular row. Note that dimensionality is also defined for a scalar (=0). Similar `Field` functions can be used to get the "global" dimensionality, shape and origin (*i.e.* for all rows in a column). However, if the dimensionality is variable (<0), the latter `shape` and `origin` functions will throw an exception.

5.4 Nested Tables

If a value (a keyword or a row in a column) is a table itself (be it direct or indirect), access to that table can be obtained by creating a `Table` object for it. This can be done using the function `table` in the `TableValue` object.

```
Table subtab = row["tab"].table();    // get table in this row
```

Thereafter this table can be accessed in the standard way; thus add rows, get and put data, *etc.*.

`Get` and `put` functions, similar to those for getting and putting scalars and arrays, can be used to get or put a table. `Get` is equal the the `table` function described above. `put` makes a reference to the table being put, which is only allowed for indirect tables.

```
Table subtab;
row["tab"].get(subtab);   // same as:  subtab = row["tab"].table()
row["tab"].put(subtab);   // put a table in this row
```

requires that `tab` is defined as an indirect table. This value now holds a reference to the table in `Table` object `subtab`.

5.5 Accessing an Entire Column

As explained in Section 5.2 [Get/Put Values], page 26, a column can be accessed on a per-row basis. However, a column can also be accessed in one go by treating it as an array of values. So a column of scalars is a vector, while a column of arrays with `N` dimensions is an array with `N+1` dimensions. Note that for this purpose arrays must have the same size in all rows, thus their full shape must have been defined before the first row is added to the

table (see Section 5.3.1 [Direct Arrays], page 27).
The `Field` functions `getColumn` and `putColumn` must be used to access an entire column. When getting a column, the receiving array must be empty (*i.e.* zero-length) or its shape must be conformant.

```
Vector<double> vec(tab.nrow());         // Create vector of right size
Field* cfpra = tab.getCol("RA");
cfpra->putColumn (vec);                 // to put a column of scalars

Array<Complex> col;
cfparr->getColumn (col);                // to get a column of arrays
```

Note that similar to getting and putting a single scalar, getting and putting a column of scalars will also promote data types if necessary.

5.6 Accessing Attributes

The `Field` object has to be used to get the attributes defined for a keyword or column. The following can be done:

`DataType dt = cfp->dataType();`
: to get the datatype (TpInt, TpDouble, *etc.*)

`String name = cfp->getName();`
: to get the name.

`String comm = cfp->comment();`
: to get the comment.

The `Table` object has to be used to get the attributes of a table. The following can be done:

`String tp = tab.getName();`
: to get the name of the table.

`String tp = tab.getType();`
: to get the table type (*i.e.* name of description for filled tables).

`uInt nrk = tab.nkey();`
: to get the number of keywords in the table.

`uInt nrc = tab.ncol();`
: to get the number of columns in the table.

5.7 Access to Keywords attached to a Field

If a keyword or column has keywords attached to it, they can be accessed using the `Field` object. Similar to the table keywords, these field keywords are treated as a row. So the `Field` function `keywords` exists to get the row of field keywords.

```
Field* cfp = tab.getCol["RA"];
Row keyrow = cfp->keywords();        // keywords attached to RA
```

Normally these two statements would be combined as:

```
Row keyrow = tab.getCol["RA"]->keywords();
```

Thereafter all functions described in the previous sections can be used to get and put data, get attributes, *etc.*.

6 Changing Table Structure

The main table and nested tables can be changed by adding, removing or renaming keywords or columns. Also keywords attached to a field can be added, removed or renamed. The way this can be done depends strongly on the `FieldDesc::VarDesc` option which could be given when defining a field (see Section 3.2 [Keywords/Columns], page 20). Note that the main table always has `FieldDesc::VarDesc` set.

- If `VarDesc` is not set, the tables in all rows of a column must have the same structure. This means that only a global change can be made.
- If `VarDesc` is set, the tables can be different in each row. This means that changes can be made to each individual table.

6.1 Changing Non-VarDesc tables

Nested tables for which the `FieldDesc::VarDesc` flag is not set, can only be changed globally (see Section 3.2 [Keywords/Columns], page 20). The tables in all rows on the column will be changed in one operation. Changing can be done using the `Field` functions:

`addKeyGlobal (const FieldDesc*);`

adds the field defined by the `FieldDesc` as a keyword to the tables in all rows. The given field description will be cloned, including possible attributes and keywords attached to the field. There is no direct constructor for a `FieldDesc` object, but one can easily be created by adding a field to a dummy scratch table.

```
TabDesc dummy(TabDesc::Scratch,"");
FieldDesc* fdp = dummy.addKey ("name",TpInt);  // define Int scalar
fdp->setComment ("...");                       // define comment
fdp->setDef ((Int)0);                          // define default
Field* cfp = table["subtab"];     // get Field of field subtab
cfp->addKeyGlobal (fdp);          // add to subtab in all rows
```

When a keyword containing a direct array is added, the shape of the array has to be defined. Similarly when adding a keyword containing a table, the description of that table has to be known.

`renameKeyGlobal (const String& newname, const String& oldname);`

renames the given keyword in the tables in all rows.

`removeKeyGlobal (const String& name);`

removes the given keyword from the tables in all rows.

In the table above the functions are shown for handling a keyword. Similar `col` functions exist to handle columns.

6.2 Changing VarDesc tables

Nested tables for which the `FieldDesc::VarDesc` flag is set, can only be changed per individual table (see Section 3.2 [Keywords/Columns], page 20). Only a table in a particular row can be changed at one time. Note that the main table also has the `FieldDesc::VarDesc` flag, so it can be changed in the very same way. Changing can be done using `Table` functions similar to the non-VarDesc tables.

`addKey (const FieldDesc*);`
: adds the field defined by the `FieldDesc` as a keyword to the table. The given field description will be cloned, including possible attributes and keywords attached to the field. There is no direct constructor for a `FieldDesc` object, but one can easily be created by adding a field to a dummy scratch table.

```
TabDesc dummy(TabDesc::Scratch,"");
FieldDesc* fdp = dummy.addKey ("name",TpInt);  // define Int scalar
fdp->setComment ("...");                       // define comment
fdp->setDef ((Int)0);                          // define default
Table t = tab[1]["subtab"].table();      // get Table in row 1
t.addKey (fdp);                          // add to that subtab
```

When a keyword containing a direct array is added, the shape of the array has to be defined. Similarly when adding a keyword containing a table, the description of that table has to be known.

`renameKey (const String& newname, const String& oldname);`
: renames the given keyword in the table.

`removeKey (const String& name);`
: removes the given keyword from the table.

In the table above the functions are shown for handling a keyword. Similar `col` functions exist to handle columns.

In addition to adding, renaming or removing a keyword or column in a table, it is also possible to do the same for a keyword belonging to a field (see Section 3.3 [Attributes], page 22 for information on keywords attached to fields). This can be done using the `Field` functions `addKey`, `renameKey` and `removeKey`, which have the same signature as the functions described above.

```
// Rename keyword OLD attached to field X1 in the main table.
Field* cfp = tab["X1"];              // get Field object
cfp->renameKey ("NEW", "OLD");       // rename keyword old to new

// Rename keyword OLD attached to field X in table subtab
// in row i of the main table.
Field* cfp = tab[i]["subtab"].table()["X"];
cfp->renameKey ("NEW", "OLD");
```

7 Selecting, Sorting and Iterating

Selecting rows from a table, sorting a table and iterating through a table result in a reference table. A reference table does not contain data itself, but references the rows in the original table. This means that changes made to a reference table will in fact be made to the original table. A reference table is handled as any other table, so select, etc. can be applied to it again.

Functions exists to combine (reference) tables having the same root table. In this way results of a select can be united, intersected, etc..

7.1 Select Expressions

A table containing selected rows of another table can be created by giving a table column expression. The resulting table can be used as all other tables.

```
Table tabsel;
tabsel = tab(tab.col["RA"] > 5  &&  tab.col["RA"] < 6);
```

The `Table` function `col` creates a special `TabExprNode` object, which can be handled by the `Table` selection operator (). The binary operators `+`, `-`, `*`, `/`, `==`, `>=`, `>`, `<=`, `<`, `!=`, `&&` and `||` and the unary operators `+`, `-` and `!` can be used. Parentheses can be used to group subexpressions. Functions (like cosine) are not provided yet, but would be rather easy to implement if necessary. Note that for strings operators `-`, `*` and `/` can not be used.. For Bools only `&&`, `||` and `!` can be used (and vice-versa).

```
tabsel = tab(tab.col["Boolcol"] ||
         tab.col["nam1"] + tab.key["nam2"] == tab.col["nam3"]);
```

For now select can only be used on scalar values, but for the future one can also think of operations on arrays via functions like minimum, average, *etc.*. Of course, all columns used in a expression must belong to the table selected on, otherwise the exception `TabInvNode` will be thrown.

Scalar '`keywords`' can also be used in a select expression; they are immediately converted to constants. However, there must always be at least one column in a select expression, otherwise an expression makes no sense.

```
tabsel = tab(tab.col["nam1"] > tab2.key["threshold"]);
```

As shown in this example, keywords may belong to another table since they are treated as constants.

The function `keyCol` can be used if it is unknown if the name of a keyword or column is given. This can be useful when parsing a select expression given at a command line interface.

7.2 Select Vector

Another, more direct, method of selecting rows from a table is the use of a vector containing row numbers to select. This can also be done using the overloaded function operator () in class `Table`. This function also creates a reference table. There are 2 versions:

Table operator() (const Vector<Int>&);

The vector must contain the row numbers to be selected from the table. The row numbers do not need to be in ascending order. The same row number can be multiply used, which has the result that that row occurs multiple times in the resulting table. If a given row number exceeds the number of rows in the table, the exception IndexError<Int> will be thrown.

Table operator() (const Vector<Bool>& mask);

The mask vector determines which rows have to be selected. If mask(i) == True, row i will be selected. An exception will be thrown if the length of the mask exceeds the number of rows. However, its length may be less.

The function using a vector of row numbers is particularly useful in combination with the Table function rowNumbers. This function returns the vector of rownumbers in a table, which can thereafter be used to create a reference table from a another table.

```
Table tab1 ("sample");
Table tab2 ("sample-related");
Table t1 = tab1(tab1.col("name") == "abc");  // select expression
Table t2 = tab2(t1.rowNumbers());            // select same rows
```

This has the result that the rows selected from tab1, will also be selected from tab2. This should only be done if the tables tab1 abd tab2 are related. The same can, of course, be done for a sorted table.

7.3 Set operations on a Table

Tables, originating from the same root, can be combined in several ways using set operations (intersection, etc.). The main purpose for this is to combine tables resulting from selects on the same main table. The set operations are available as overloaded operators.

Table operator& (Table& this, Table& that);

makes the intersection of 2 tables.

Table operator| (Table& this, Table& that);

makes the union of 2 tables.

Table operator- (Table& this, Table& that);

subtracts table that from this.

Table operator^ (Table& this, Table& that);

makes the exclusive or of 2 tables. It is equal to subtracting the intersection from the union.

Table operator! (Table&);

makes the complement of the table; *i.e.* take all rows from the root table which are not in this table. It is equal to subtracting the table from the root (or xor-ing the table with the root).

```
Table tab("sample");                 // root table
Table t1 = tab(tab.col("c1") > 1);  // select some rows
Table t2 = tab(tab.col("c2") > 10);
```

```
Table t3 = t1 & t2;                 // intersect the select results
Table t4 = !t1;                     // same as  tab.col("c1") <= 1
                                    // and as   tab - t1
```

7.4 Sorting a Table

A table can be sorted in ascending or descending order on any column containing a scalar value. It can also be sorted on multiple columns in a mix of ascending and descending order. The sort functions comes in 3 flavours:

`Table sort (const String& name, const Int ord, const Int opt);`
: This is the simplest version allowing a sort on a single column. The optional 2nd argument gives the sort order and defaults to `Sort::Ascending`. Options can be given in the optional 3rd argument.

  `Sort::HeapSort (default)`
  `Sort::QuickSort`
  `Sort::InsSort`
  : Only one of these options can be given. It determines which sort algorithm (heapsort, quicksort or insertion sort) will be used in the `Sort` class. Note that insertion sort is usually very slow, but it is the only stable algorithm. *I.e.* it keeps the original order of equal sort keys.

  `Sort::NoDuplicates`
  : Skip data with duplicate sort keys.

`Table sort (const Vector<String>& names, const Int ord, const Int opt);`
: This allows a sort on multiple columns, all in the same order. The most important sort key has to be specified first. The optional arguments `ord` and `opt` are the same as above.

`Table sort (const Vector<String>&, const Vector<Int>& ord, const Int opt);`
: This allows a sort on multiple columns with the order being specified per column. Of course, both vectors must have the same length. The argument `opt` is the same as above.

Similar to select, the table resulting from a sort only contains references to the rows in the original table.

```
Table tab("XYZ");                     // open table XYZ
Vector<String> keys(2);
Vector<Int> order(2);
keys(0)  = "OBSERVER";
order(0) = Sort::Ascending;
keys(1)  = "DATE";
order(1) = Sort::Descending;
Table tabs = tab.sort (keys, order);  // sort on observer,date
Row row;
for (uInt i=0; i<tabs.nrow(); i++) {
    row = tabs[i];                    // loop through sorted table
      ...
```

```
}
Vector<double> dat;
tabs.getCol("RA")->getColumn(dat);    // get RA in sorted order
```

7.5 Command String Syntax

Apart from the ability to select and sort rows using the C++ functions and overloaded operators described in the previous sections, it is also possible to select and sort using a command string. This is particularly useful for an interface to the end-user.

The function `tabCommand` parses and executes an SQL-like table select/sort command string. It returns the table created. This (global) function is defined in the header file `TabParse.h`. Its signature is:

```
Table tabCommand (const String& command);
```

The syntax of the command string is:
`select` table_in `where` expression `orderby` columns `giving` table_out

Rows in the table designated by 'table_in', are selected and/or sorted and the result is stored in the table designated by 'table_out'. Selection can be done by specifying the select-part "`where` expression". Sorting can be done by specifying the sort-part "`orderby` columns". Either the select-part, sort-part or both has to be given.
In case of a syntax error, `tabCommand` throws the exception `TabParseError`.

```
select Table.data where col1>10 giving Table.1
select Table.1 orderby col1, col2 giving Table.2
select Table.data where col1>10 orderby col1,col2 giving Table.1
```

The first example selects all rows from table `Table.data` where column `col1` is greater than 10.
The second example sorts the result of the first in order of columns `col1` and `col2`.
The third example combines the previous commands in one command.

As shown in the examples, multiple sort keys can be given (separated by commas). As with the C++ table select functions and overloaded operators, an expression can be of any complexity. A select has to be done on columns, so at least one column name has to used in the select expression. However, it is possible to use a keyword as shown in the following example, which is similar to the example given in Section 7.1 [Select Expressions], page 33. In this example column `nam1` is compared with keyword `threshold`.

```
select Table.data where nam1>threshold giving Table.out
```

It is not possible (yet) to join tables. This means that columns used in a select-expression can only be taken from one table. It is, however, possible to use keywords from multiple tables. This can be done by naming all tables used as input tables, defining a shorthand for them and qualifying the keyword names with the shorthand. The table containing the tested columns has to be the first table specified.

```
select table.nm1 t1, table.nm2 t2 where t1.col1>t2.threshold giving table.3
```

In the above example column `col1` in table `table.nm1` is compared with keyword `threshold` in table `table.nm2`. A few remarks:

1. The string "table.nm1 t1" defines shorthand t1 for table name table.nm1. The shorthand has to be an alphanumeric string. It defaults to the table name, so it needs to be specified if the table name contains special characters (like the . in this example).
2. Tables names have to be separated by commas.
3. The first table is the main table, *i.e.* it is the table from which rows will be selected. If a sort-part is given, the first table is also the table being sorted and the specified sort columns have to be columns in this table.
4. The column and keyword names used in the select-expression do not need to be qualified with the table shorthand, if they are part of the first table. So in this example col could have been used instead of t1.col. This means that in practice qualification is seldom needed.

7.6 Iterating through a Table

Each step in an iteration through a table returns a table referencing the rows in which the iteration keys have equal values. The iteration keys have to be columns containing a scalar value. Iteration can be done in any order.

Multiple iteration objects can be constructed for the same table allowing to iterate simultaneously in different ways through the same table . A TableIterator object has to be constructed to perform iteration. The constructor comes in 3 flavours:

TableIterator (Table&, const String& name, const Int ord);

This is the simplest form which allows to iterate on a single column. The name of the column has to be specified in the 2nd argument. The optional 3rd argument specifies the iteration order, which can be one of the following options.

TableIterator::DontCare (default)

The iteration order is not important as long as the rows with equal iteration keys are returned in one table. This option allows the table system to optimize as much as possible.

TableIterator::Ascending

Iterate through the table in ascending order. This may involve an implicit sort operation.

TableIterator::Descending

Iterate through the table in descending order. This may involve an implicit sort operation.

TableIterator (Table&, const Vector<String>& name, const Int ord);

This allows to iterate on multiple columns, all in the same order defined by the optional 3rd argument.

TableIterator (Table&, const Vector<String>&, const Vector<Int>& ord);

This constructor is similar to the previous one, but in the 3rd argument the iteration order can be specified per column.

An iteration itself requires only 3 TableIterator functions.

`Bool pastEnd();`
: is the iteration guard. It returns `True` if the entire table has been processed.

`Table table();`
: returns a `Table` object for the current iteration position.

`void next();`
: advances the iterator to the next position.

```
Table tab("XYZ");                  // open table XYZ
Vector<String> keys(2);
keys(0) = "TIME";
keys(1) = "BASELINE";
TableIterator iter(tab, keys);     // construct iterator
while (!iter.pastEnd()) {
    Table t = iter.table();        // get rows with equal time,baseline
      ...
    iter.next();                   // advance to next iteration group
}
```

8 Table Vectors

The templated class `TableVector` allows to handle data in a column or a row of a table as a vector. It refers to the values in the column or row mentioned when constructing the table vector. A `TableVector` can be handled in the very same way as a `Vector` object. Thus it can be indexed, added, etc.. Slicing is the only `Vector` operation which has not been defined for table vectors (yet).

Note that an assigment to a table vector has the effect of putting data in the table values referred to by the table vector.

There are 3 different kind of table vectors, but apart from constructing them there is no difference. The only special thing is the `setRow` function for table row vectors to indicate the table row to be used.

8.1 Table Column Vector

A table column vector is a table vector referring a column in a table. It can be constructed using:

`TableVector<T> (Table&, const String& columnName);`

or

`TableVector<T> (Table&, const Int columnIndex);`

The table column can contain scalar values or arrays. In case of scalar values, the table vector type does not have to match the data type of the column (although it is more efficient if it does). However, data types can only be promoted. For example, a `TableVector<double>` can be constructed for a table column containing `float` values, but not a `TableVector<Int>`.

In case a column contains arrays, data type promotion is not possible. Furthermore the shape of all arrays in the column have to be defined and have to be the same. Note that if, for example, a column contains vectors, the result is a table vector of vectors and not a matrix.

```
Table tab("XYZ");
TableVector<Int> tvec(tab, "COL1");    // table column vector COL1
tvec = 0;                              // set entire column to 0
```

8.2 Table Row Vector

A table row vector is a table vector referring values in a row in a table. When constructing a table row vector, the names of the columns in the rows have to be specified. It can be constructed using:

`TableVector<T> (Table&, const Vector<String>& columnNames);`

or

`TableVector<T> (Table&, const Vector<Int>& columnIndices);`

The rules regarding scalars/arrays mentioned above, also apply to table row vectors. Thus it is possible to have a table row vector of values with different data types by constructing a `TableVector` object with an appropriate type.

A table row vector does not refer to a particular row in the table. There is the special `TableVector` function `setRow` to set a specific table row.

```
Table tab("XYZ");
Vector<String> nam(3);
nam(0) = "U";
nam(1) = "V";
nam(2) = "W";
TableVector<double> tvec(tab, nam);    // table row vector
for (Int i=0; i<tab.nrow(); i++) {    // loop through all rows
    tvec.setRow(i);
    tvec = 0;                          // initialize U,V,W
}
```

8.3 Table Memory Vector

In order to store the intermediate results from operations on table vectors (*e.g.* when adding two table vectors), a special kind of table vector, the table memory vector, is needed. Currently it uses a `Vector` object, but in the future it may be needed to map it to a file for really large table vectors.

Table memory vectors will be constructed automatically when needed. It is, however, also possible to construct them manually using the constructor
`TableVector (const uInt leng);`
which constructs an empty table memory vector of length `leng`.

To allow transparent use of `Vector` objects in `TableVector` expressions, there is also the table memory vector constructor
`TableVector (const Vector<T>&);` This constructor uses the `Vector` copy constructor and, therefore, has reference semantics.

8.4 Table Vector Functions

Table vectors behave exactly the same way as normal vectors. So the following functions are defined for them:

1. Copy constructor having reference semantics.
2. Assignment operator having copy semantics.
3. Index operator ().
4. Functions to get the shape of the vector. Note that the origin of a table vector is always 0.
5. Mathematical operators and functions like +=, *, `sin`. They are defined in the header file `TabVecMath.h`.
6. Relational operators like ==, `<`. They are defined in the header file `TabVecLogic.h`.

Futhermore a special function `makeVector` exists to copy a `TableVector` object to a `Vector` object.

9 Detailed Implementation

This chapter is intended for users who need to implement table classes derived from the base table classes. It describes rather detailed which member functions must or can be implemented in the derived classes.

9.1 Implementation Overview

Chapter 2 [Table Overview], page 16 described the main classes of the table system. Derived from these classes are the classes `FillTable` and `FillField` to implement the filled tables.

Virtual table classes can be derived from the basic table classes `BaseTable` and `Field`. It is, however, also possible to derive them from `FillTable` and `FillField`, since they offer some more functionality. For instance, they offer a table and field description and a mapping of keyword and column names to `FillField` objects using the template class `OrderedMap`.

A tentative virtual table class `VirtualTable` and its associated classes `VirtualFld...` have been checked in into the system and may serve as an example. They are derived from `FillTable` and its associated `FillFld...` classes. Note that this is only one (simple) way of implementing a virtual table. Other kind of virtual tables will probably need other things and may need to be derived from `BaseTable` and `Field`.

The following section in this chapter describe the base classes `BaseTable` and `Field` which form the foundation of the general table system. It also describes the important classes `Row` and `TableValue`, which glue the `BaseTable` and `Field` classes together to allow access to the data in the table fields. `TableValue` is quite simple, but `Row` is somewhat more complicated for efficiency reasons.

The section thereafter describes the derived classes used for implementing the filled tables.

9.2 Base Classes

9.2.1 BaseTable

A class derived from BaseTable has to implement the virtual functions defined in BaseTable. There are 2 types of virtual functions:

- Pure virtual functions which have to be implemented in the derived class, otherwise it is also an abstract class which cannot be instantiated. Most virtual functions in `BaseTable` are pure virtual.
- Virtual functions having a default implementation in `BaseTable`. Often this implementation is throwing an invalid action exception. This is, for example, done for the function `addRow`; so if the derived class does not support addition of rows the default implementation is sufficient, otherwise `addRow` has to be implemented. Note that 4 virtual functions have been defined for the sake of `RefTable` (the result of a select or sort). These functions are `root`, `rowOrder` and `rowStorage`. They do not have to implemented, since their default implementations are always sufficient.

Apart from the virtual functions (take care of the **virtual destructor**!!), the constructor(s) and the `cleanup` function also have to be implemented in derived classes. Since `BaseTable` only has a default constructor, it is not needed to call it explicitly in the constructors of derived classes.
The `Table` class forms a thin layer between the table classes and the user, so there must be a way to construct a `Table` object for each table type. `Table` has the constructor

```
Table (BaseTable*);
```

which can in general be used. However, for some often used table types it may be useful to have special `Table` constructors to make life easier for the user. This is, for instance, done for filled tables.

The header file `BaseTable.h` should be examined for the exact details. The comments in it tell the default implementation of non-pure virtual functions. Note that the entire interface has to be exmanined, not only the public. Sometimes a public function may be using a virtual private function.

It may also be a good idea to look at `FillTable.h` to see how the class for filled tables is defined and implemented.

9.2.2 Field

In principle this abstract base class is designed in the same way as `BaseTable`, so pure virtual functions has to be implemented non-pure virtual functions can be implemented. However, this class is much bigger and more complicated. A complicating factor is, for instance, that it has to deal with the different data types of the fields, which gives rise to a lot of similar functions (*e.g.* a `get` for Bool, Int, double, *etc.*.

There are many public non-virtual functions translating into public functions. There are, for example, 5 versions of `setDim`, but 4 of them translate into the first virtual version, so only that version of `setDim` has to be implemented in derived classes (provided `setDim` is supported).

The `Field` functions can be divided into the following groups:

1. The constructor, which has to be called explicitly.
2. The destructor, which is virtual.
3. To handle keywords attached to the field defined by Field.
4. To get attributes of a field.
5. To handle the data in a particular row of the field.
6. To handle an entire column.
7. Miscellaneous.

9.2.2.1 Field Constructor

The `Field` object contains several basic variables.

`BaseTable* btp`
: This is the link to the table the field belongs to.

`DataType dtype`
: The data type of the field as defined in `DataType.h`.

`int dtsize`
: The size of the data type on the machine in use. The size of a string is the size of the String object.

`Int nrdim`
: The dimensionality of the values in all rows of the field. Note that 0 inidicates a scalar and -1 indicates an array with a yet undefined dimensionality. A field containing tables always has dimensionality 1.

`Bool swkey`
: `True` means that the field is a keyword, `False` a column.

`TableValue fld`
: This is defined for effiency reasons. It is used by class `Row` to be able to return a reference (`TableValue&`).

There is one constructor for the `Field` object, which has to called when constructing objects derived from it to initialize the variables described above. Its signature is:
`Field (BaseTable*, const DataType, const Int nrdim, const Bool swkey);`

9.2.2.2 Field Destructor

The destructor of the `Field` is virtual, so a destructor has to be defined in the derived classes. Field is **not** derived from `Cleanup`, so no cleanup function has to be defined. The reason for not using `Cleanup` is that all `Field` objects will be deleted by the class derived from `BaseTable`, so no memory leaks will occur for them. Using `Cleanup` could result in deleting the same `Field` object twice.

9.2.2.3 Keywords attached to a Field

The functions to handle the keywords attached to a field are similar to the corresponding functions in `BaseTable`.

`nkey, getKey, getKeyIndex, keywords`
: are pure virtual functions.

`operator[]`
: is not virtual. It is similar to `getkey` and implemented as such.

`addKey, removeKey, renameKey`
: throw an `TabInvAct` exception, indicating that the action is invalid.

If they need to be supported, class `FillField` shows how it can be done.

9.2.2.4 Field Attributes

There are a few functions dealing with getting the attributes of a Field. Most of these functions are pure virtual and has to be implemented in the derived class, while the remaining ones are not virtual at all.

`getName`
: gets the name of the field. It is pure virtual.

`comment`
: gets the comment describing the field. It is pure virtual.

`dataType`
: gets the data type. It is not virtual.

`isKeyword`
: tells if the field is a keyword. It is not virtual.

`direct, indirect`
: tells if a field (an array or table) is direct or indirect. It is pure virtual.

9.2.2.5 Field Row Data

The programmer uses classes `Row` and `TableValue` to get access to the data in a specific row. However, these classes only form an interface to the appropriate `Field` functions which get and put the data and set the shape of an array. All these functions get the row number as the first argument.

Getting and putting data needs to be done for many data types, which requires all instances of a function in all derived classes, otherwise the compiler complains about hiding functions. To avoid this, `Field` converts all get and put functions to a few general functions using `void*` for the data buffer. In this way a derived class has to deal with only a few functions, but it requires a cast to the appropriate type. The following virtual functions have to be implemented in the derived classes. The default implementation of these functions throw an `TabInvAct` exception.

`initialize`
: to initialize the value in one or more rows to its default value.

`bufGet, bufPut`
: to get and put a scalar value. Conversion of scalar values to other data types is handled by `Field`, so the derived class does need to worry about that. Note that `bufGet` needs to return a pointer to the data value, so that value should not be stored in an automatic variable because that would be deleted before control is returned to the calling function.
Note that the functions `get` and `put` are optimized by inlining the cases without data type conversion. For the other cases `getConv` and `putConv` are called. All these functions use `bufGet` and `bufPut` to access the actual data.

`getArray, putArray`
: to get and put an array of values. Data type conversion is not supported for arrays, which means that the functions has to check if the given data type matches the field data type. If not, the function `throwInvDT` can be called to throw the `TabInvDT` exception.

`getSlice, putSlice`
: to get or put a slice of an array. As with arrays, the data type needs to be checked. Furthermore a slice can be given implicitly using the _ notation (see class `Vector`), which **must** be converted to an explicit slice. This can be done using the function `setbti`, which sets the explicit values in the *blc*, *trc* and *inc* vectors.

`table, getTable, putTable`
: to get access to a nested table. Only `table` and `putTable` are virtual functions, while `getTable` is not (it uses `table`). `putTable` must test if the nested table in the column is indirect, since direct tables cannot be put (otherwise another

type of table could be put). The default implementation of these functions is throwing a `TabInvAct` exception.

`setRowDim`
: to set the shape and origin of an array (or table). The function `combCheckDim` is very useful for this purpose. It combines the given shape and origin with the possibly already partly defined global shape and origin for all values in the column (see Section 9.2.2.6 [Field Column], page 45 and it checks if the shape is fully defined.

`rowNdim, rowShape, rowOrigin`
: to get the dimensionality, shape and origin of an array. By default `rowNdim` returns `nrdim`, which is fine for scalars and tables, which have a fixed dimensionality. However, in general it needs to be implemented for arrays.
The other 2 functions throw by default a `TabInvAct` exception..

9.2.2.6 Field Column Data

While the functions in the previous section handle one row in a column, the functions described in this section handle an entire column.

`getColScalar, putColScalar`
: Getting a column of data is handled by several functions. The only virtual function is `getColScalar`.
The function `getColumn` is the basic function doing all checks. The actual work is done by `getColArray` for arrays and by `getColConv` for scalars. They use the `get` and `put` functions described in the previous section. The case of getting a column of scalars without data type conversion is handled separately in the virtual function `getColScalar`, because it can potentially be handled more efficient in a derived class. By default `getColScalar` is implemented using `getColConv`.
The same is true for putting a column of scalars, where `putColScalar` is the virtual function.

`setDim`
: sets the global shape and origin of all values in the column. It is not necessary to define the shape completely yet; dimensions can be set to zero. The function `combDim` is useful for this purpose. It checks the dimensionality and the shape anhd combines it with the possibly already defined shape.
By default `setDim` throws a `TabInvAct` exception.

`ndim, shape, origin`
: return the dimensionality, shape and origin. By default `ndim` returns `nrdim`, while the other 2 throw a `TabInvAct` exception.

`getDataPtr, delDataPtr`
: These functions are somewhat special. They are intended for the sort operation on a table.
The virtual function `getDataPtr` returns the address of the data of a column. The default implementation is to allocate storage and read the data into it, but derived classes can be smarter. For instance, the filled classes can immediately

return the address.
The virtual function `delDataPtr` deletes the memory possibly allocated by `getDataPtr`.

`addKeyGlobal, renameKeyGlobal, removeKeyGlobal`
`addColGlobal, renameColGlobal, removeColGlobal`
These functions are only for a column containing direct tables. They add a keyword or column to all the tables in the column. Their default implementation is throwing a `TabInvAct` exception.

9.2.2.7 Field Miscellaneous

Several functions in class `Field` have not been mentioned. All of them are not virtual, except the IO functions `getFil` and `putFil`. These functions are usually not appropriate for derived tables, except for filled tables. Their default implementation is throwing a `TabInvAct` exception.

Furthermore the virtual functions `allValueBuf` and `delValueBuf` are defined. They allocate, delete, resp. a buffer to hold a value for the field. These functions are used by class `BaseTableIterator` to have a buffer to hold values of a field in a `Field` object. Currently these functions are only implemented by the `FillFldSca` classes.

9.2.3 Row

A reference to a `Row` object is returned by some functions in the classes `BaseTable` and `Field`, or to be more precise, by classes derived from them. Since this operation is done quite often, it should be optimized. The way this is done by `FillTable` is defining 2 `Row` objects (one for the keywords and one for the columns) and returning a reference to them after setting the correct row number in them using the `Row` function `setRow`. Similarly class `FillField` has a `Row` object for keywords attached to a field.

A row is the set of keywords or the set of columns at a particular row. This means that the `Row` object has to know which elements are part of the set. There are 4 functions doing this which should be called in the constructor of the classes derived from `BaseTable` and `Field`.

`setTabKeyPtr (BaseTable*);`
defines that the `Row` object represents the keywords of the given table.

`setTabColPtr (BaseTable*);`
defines that the `Row` object represents a row in the columns of the given table.

`setFieldPtr (Field*);`
defines that the `Row` object represents the keywords attached to the given field.

`setSetPtr (FieldSet*);`
is a special one for the filled tables. Since filled tables use the class `FieldSet` for the set of keywords and columns, `Row` could be optimized by taking advantage of that.

The constructors of classes `FillTable` and `FillField` and the functions `addRow`, `operator[]` and `keywords` in them show how it is implemented for the filled tables. For other table types a similar scheme should be used.

9.2.4 TableValue

`TableValue` is a very simple class. All functions in it are inlined and simply consist of calling the appropriate `Field` function while passing the row number.

A reference to a `TableValue` object is returned by the index operator [] in class `Row`. Since this operation will be done very often, it has been optimized by defining a `TableValue` object in class `Field` and returning a reference to it. `Row` fills in the correct row number using the `Field` function `setTableValueRow`.

9.3 Filled Table Classes

– to be written –

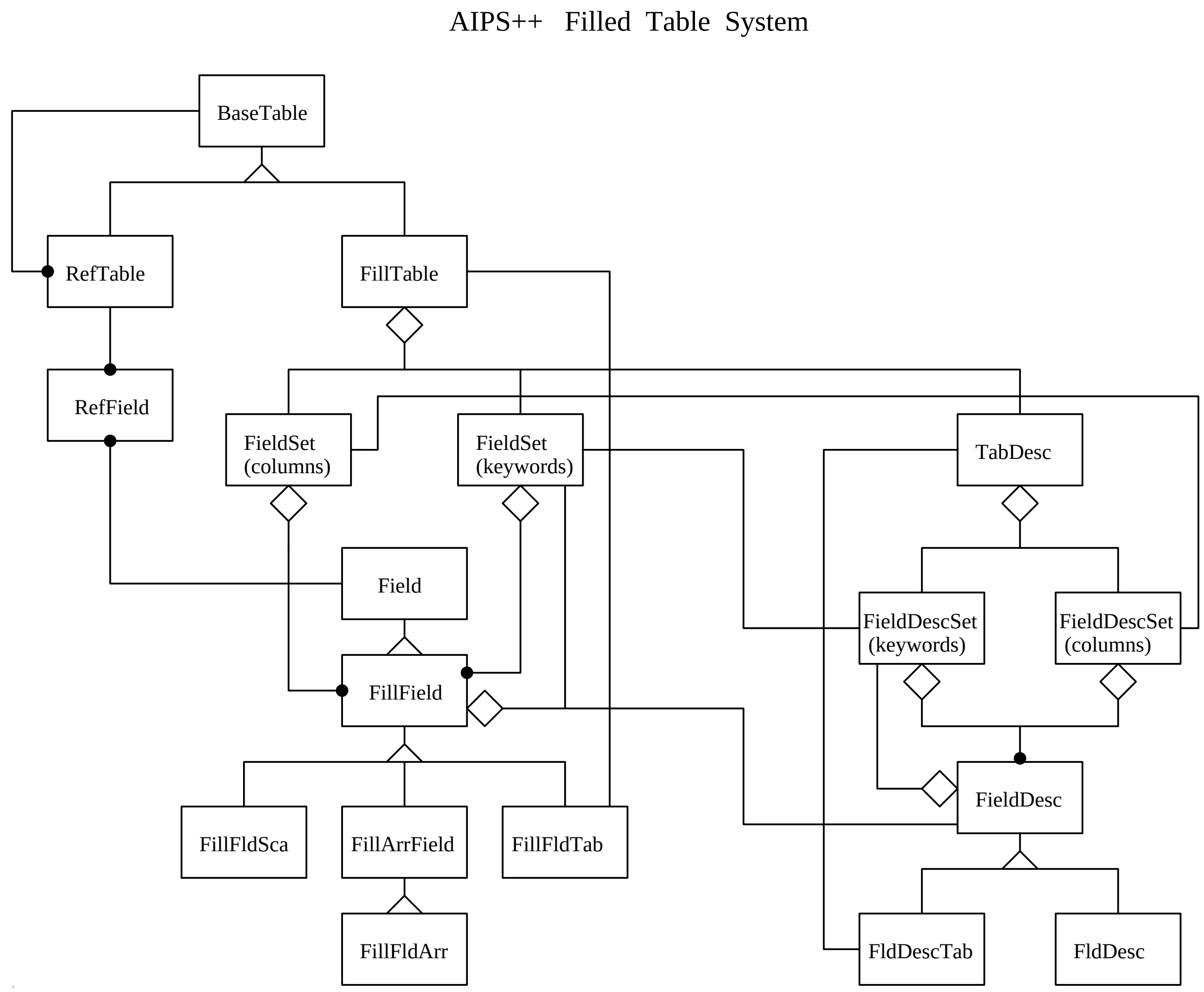

Figure TABCLASS2 – Filled Table Classes.

Appendix A Examples

This example shows how a table description is created and stored in a file. The filename used will be the name of the description followed by the suffix `.tabdesc`, thus resulting in this example in `Example.tabdesc`.

It shows how keywords and columns are defined. It also shows how the returned `FieldDesc*` can be used to attach a keyword to a table keyword and to set a default value.

```
// Define a table description.
//

#include <aips/TabDesc.h>
#include <aips/FieldDesc.h>

main()
{
//
// Create an empty description and name it Example.
//
    TabDesc td("Example", TabDesc::New);
//
// Add several keywords to it.
// Date and name are strings.
// Freq is a 1-dim array with the observed frequencies.
// The associated unit is stored with it. It defaults to Mhz.
//
    td.addKey ("date", TpString);
    td.addKey ("name", TpString);
    FieldDesc* fd = td.addKey ("freq", 1, TpFloat);
    fd = fd->addKey ("unit", TpString);
    fd->setDef ("Mhz");
//
// Now add some columns.
//
    td.addCol ("U", TpDouble);
    td.addCol ("V", TpDouble);
    td.addCol ("time", TpFloat);
    td.addCol ("baseline", TpInt);
    td.addCol ("vis", 1, TpComplex);
//
// On exit the TabDesc object is deleted.
// This executes its destructor, which will store the description
// in a file.
//
}
```

This example shows how a filled table is created using the description from the previous example. The file in which the table is stored is the table name, in this case `table.name`.

Note that the `Row` and `TableValue` objects are used extensively to put the data values. It can also been seen in the line where the value 'Khz' is put for keyword `unit`, that in C++ one statement can consist of multiple statements. That has the effect that several objects may only be used implicitly.

```
// Create and fill a table.
//

#include <aips/Table.h>
#include <aips/Row.h>
#include <aips/TableValue.h>
#include <aips/Field.h>
#include <aips/Vector.h>

main()
{
    uInt nrfreq = 32;
    uInt nrrow  = 1000;
//
// Create the table using the Example description.
//
    Table tab("table.name", "Example", Table::New);
//
// First put the values for the keywords.
// The set of keywords is handled as a row.
// A row can be indexed by the field name giving a TableValue.
//
    Row keyrow = tab.keywords();
    keyrow["date"].put ("05-28-93");
    keyrow["name"].put ("Ger van Diepen");
//
// Freq is a 1-dim array, whose length is undefined.
// Its length has to be defined, before data can be put.
//
    Vector<float> freqval(nrfreq);
    for (Int i=0; i<nrfreq; i++) {
        freqval(i) = i*100;
    }
    TableValue fld = keyrow["freq"];
    fld.setDim (nrfreq);
    fld.put (freqval);
//
// The unit is Khz, so put that as well.
// The statement used is equal to:
//    Field* cfp = tab.getKey("freq");
//    Row frow = cfp->keywords();
//    TableValue ffld = frow["unit"];
```

```
//    ffld.put("Khz");
//
    tab.getKey("freq")->keywords()["unit"].put ("Khz");
//
// Now put the data into all the rows.
// Get the column index of the columns, since indexing by
// name for each row is a bit expensive.
//
    Row row;
    Int ucol   = tab.getColIndex("U");
    Int vcol   = tab.getColIndex("V");
    Int timcol = tab.getColIndex("time");
    Int bascol = tab.getColIndex("baseline");
    Int viscol = tab.getColIndex("vis");
    for (i=0; i<nrrow; i++) {
        row = tab.addRow();
        row[ucol].put(Uvalue);
        row[vcol].put(Vvalue);
        row[timcol].put(timevalue);
        row[bascol].put(baselinevalue);
        row[viscol].setdim(nrfreq);
        row[viscol].put(visvector);
    }
//
// Table destructor will write the data and close the file.
//
}
```

This example shows how the table created in the previous example is read back. A select expression is used to select specific rows from the table. The result of the select is sorted before being read.

It also shows how an entire column of vectors can be read into a matrix using the `Field` column functions.

```
// Access the data in an existing table.
// Select and sort the data.

#include <aips/Table.h>
#include <aips/Row.h>
#include <aips/TableValue.h>
#include <aips/Field.h>
#include <aips/TabExprNode.h>  // for select expression
#include <aips/Vector.h>
#include <aips/Matrix.h>
#include <aips/ArrayIO.h>      // for cout << visvector

main()
{
// Open an existing table.
// Select the rows for which U*U + V*V < 1.
// Sort the result on time (default is ascending).
//
    Table tab("table.name");
    Table seltab = tab(tab.col("U") * tab.col("U") +
                       tab.col("V") * tab.col("V") < 1);
    Table sortab = seltab.sort ("time");
//
// A sort on multiple columns is also possible; e.g.
//
    Vector<String> nam(2);
    Vector<Int>    ord(2);
    nam(0) = "time";
    ord(0) = Sort::Descending;
    nam(1) = "baseline";
    ord(1) = Sort::Ascending;
    Table sortab2 = sortab.sort (nam, ord);
//
// Now show the visibilities in all selected rows.
//
    Row row;
    Vector<Complex> visvector;
    for (Int i=0; i<sortab.nrow(); i++) {
        row = sortab[i];
        row["vis"].get(visvector);
        cout << "Row " << i << ": " << visvector << endl;
    }
//
// Alternatively the entire column could be read as a matrix.
```

```
//
    Matrix<Complex> vismatrix;
    sortab.getCol("vis")->getColumn (vismatrix);
    cout << vismatrix << endl;
}
```

This example shows the use of a table vector. Note that the assignment to the `timetv` table vector has the effect that new values are put in the table for column `time`. Therefore the table is opened with option `Table::Update`.

```
// Example of use of table vectors.

#include <aips/Table.h>
#include <aips/TableVector.h>
#include <aips/TabVecMath.h>   // for math on TableVector

main()
{
// Open the table; allow for updates.
// Create a table vector for column time.
//
    Table tab("table.name", Table::Update);
    TableVector<float> timetv(tab, "time");
//
// Manipulate the vector (double the values).
//
    timetv = 2*timetv;
//
// Allocate a memory vector with the same length.
// Initialize to zero.
// Copy every other element.
//
    uInt nr = timetv.nelements();
    TableVector<float> tmp(nr);
    tmp = 0;
    for (Int i=0; i+=2; i<nr) {
        tmp(i) = timetv(i);
    }
//
// Slicing is not supported yet (it is in class Vector).
// Otherwise the above loop could have been replaced by:
//
//    tmp(_(0,nr-1,2) = timetv(_(0,nr-1,2));
}
```

This example shows the use of a table iterator. The iteration is on column `time` in descending order. So the first iteration step returns a subtable containing all rows with time=N, the next iteration step returns a subtable containing all rows with time=N-1, etc..

```
// Example of use of table iterators.

#include <aips/Table.h>
#include <aips/TableIter.h>
#include <aips/Vector.h>

main()
{
// Open the table.
// Iterate through the table on column time in descending order.
//
    Table tab("table.name", Table::Update);
    Vector<String> itvec(1);
    itvec(0) = "time";
    TableIterator iter(tab, itvec, TableIterator::Descending);
//
// Now iterate through the table.
// Each step will return a subtable, containing all rows for
// which the time is equal.
//
    while (!iter.pastEnd()) {
        Table t = iter.table();
        //        ...
        // do something with this table ...
        //        ...
        t.next();
    }
}
```

Class Index

Function Index

(Index is empty)

Figure Index

Concept Index

D

T

Table of Contents