Radio data archiving system

Knapic, C., Zanichelli, A., Dovgan, E., Nanni, M., Stagni, M., et al.

C. Knapic, A. Zanichelli, E. Dovgan, M. Nanni, M. Stagni, S. Righini, M. Sponza, F. Bedosti, A. Orlati, R. Smareglia, "Radio data archiving system," Proc. SPIE 9913, Software and Cyberinfrastructure for Astronomy IV, 99132D (26 July 2016); doi: 10.1117/12.2232603

SPIE.

Event: SPIE Astronomical Telescopes + Instrumentation, 2016, Edinburgh, United Kingdom

Radio Data Archiving System

Knapic C.[a], Zanichelli A.[b], Dovgan E.[a,c], Nanni M.[b], Stagni M.[b], Righini S.[b], Sponza M.[a], Bedosti F.[b], Orlati A.[b], and Smareglia R.[a]

[a]National Institute of Astrophysics – Astronomical Observatory of Trieste, via G.B. Tiepolo 11, Trieste, Italy
[b]National Institute of Astrophysics – Radio Astronomy Institute, via P. Gobetti 101, Bologna, Italy
[c]Jožef Stefan Institute, Jamova cesta 39, Ljubljana, Slovenia

ABSTRACT

Radio Astronomical Data models are becoming very complex since the huge possible range of instrumental configurations available with the modern Radio Telescopes. What in the past was the last frontiers of data formats in terms of efficiency and flexibility is now evolving with new strategies and methodologies enabling the persistence of a very complex, hierarchical and multi-purpose information. Such an evolution of data models and data formats require new data archiving techniques in order to guarantee data preservation following the directives of Open Archival Information System and the International Virtual Observatory Alliance for data sharing and publication. Currently, various formats (FITS, MBFITS, VLBI's XML description files and ancillary files) of data acquired with the Medicina and Noto Radio Telescopes can be stored and handled by a common Radio Archive, that is planned to be released to the (inter)national community by the end of 2016. This state-of-the-art archiving system for radio astronomical data aims at delegating as much as possible to the software setting how and where the descriptors (metadata) are saved, while the users perform user-friendly queries translated by the web interface into complex interrogations on the database to retrieve data. In such a way, the Archive is ready to be Virtual Observatory compliant and as much as possible user-friendly.

Keywords: Radio Astronomy, Archive, Radio Data Model

1. INTRODUCTION

Recently, Medicina and Noto Radio telescopes and VLBI Italian observation (including Sardinia Radio Telescope data) have been equiped with a new archiving system capable to store all the private and public data acquired with those facilities and export them to the Astronomical Community, using a common web interface and following an official data policy. This is a joint collaboration between the Radio Astronomy Institute and the Italian Astronomical Archives (IA2) infrastructure at the Astronomical Observatory of Trieste that aims at providing the (inter)national community with a state-of-the-art archive for radio astronomical data and will in the near future be provided also with Virtual Observatory compliant services to increase the interoperability of data. The IRA radio telescopes at Medicina and Noto can observe in Single-Dish (SD) or VLBI mode. In particular, with the advent of the new Sardinia Radio Telescope it will be possible to realize a fully Italian VLBI array (hereafter VLBI-IT) by using the three antennas in a coordinated manner and applying software correlation for the pre-processing phase. Data coming from such pre-processed VLBI-IT observations will be stored in the same database used for SD data. The Archive must thus be able to host and handle two kinds of data: those coming from VLBI-IT observations and those coming from SD ones, with a variety of formats that will be described in the following sections.

Modern Single Dish acquisition modes are described by a vast number of possible instrumental setup parameters. This required the design and implementation of a Radio Data Model able to store all the information

Further author information: (Send correspondence to Cristina Knapic)
E-mail: knapic@oats.inaf.it, Telephone: +39 040 3199 276

Software and Cyberinfrastructure for Astronomy IV, edited by Gianluca Chiozzi, Juan C. Guzman, Proc. of SPIE Vol. 9913, 99132D · © 2016 SPIE
CCC code: 0277-786X/16/$18 · doi: 10.1117/12.2232603

needed to fully characterize the various observations. The developed radio data model has been built on top of the data/metadata structure defined in the MBFITS standard for the Atacama Pathfinder Experiment (APEX) and is capable to handle radio data written in FITS format as well, as described in Sect. 2. At the Medicina, Noto and SRT sites –for VLBI-IT acquisition mode– a customized XML summary file stores the main configuration parameters for the interferometric observations and is a subset of the previously mentioned Radio data model. The archiving system itself is equipped with an internal data model in order to handle transparently all the various radio raw data formats mentioned above. Additional observation–related information such as schedules and night logs are archived together with the data.

The Radio Archive users can perform dedicated queries through the Radio Archive web interface. SD and VLBI-IT observations share a generic subset of parameters. A more specialized query form specific for each instrumental mode (SD or VLBI-IT) in terms of selection parameters and query outputs is also available.

2. RADIO RAW DATA MODELS AND METADATA

2.1 MBFITS

Initially MBFITS was developed to define a new, FITS-based data format for multifeed receivers to be used at the IRAM 30m and APEX telescopes.[1] Later on, changes in the format structure and the addition of keywords needed to accommodate single-dish configurations (particularly multiple beam observing and multiple frontend/backend combinations) have been done.[2] It can be used for all single-dish bolometer and heterodyne observations including multiple frontend/backend combinations and array receivers. The MBFITS hierarchical grouping directory is defined as follows:

- Main directory name according to the OBSDATE and PROJID keywords. Inside this main directory there are the files for the scan-level tables:
  - the grouping table file: GROUPING.fits
  - the scan table file: SCAN.fits
  - the FEBEPAR table files for each FEBE (instrumental frontend/backend) combination: <FEBE name>-FEBEPAR.fits
- The actual data are stored in sub-directories per subscan named according to the subscan number. Each sub-directory contains the following types of member files:
  - the MONITOR table file: MONITOR.fits
  - the ARRAYDATA table files for each FEBE combination and baseband:
    - <FEBE name>-ARRAYDATA-<Baseband number>.fits
    - the DATAPAR table files for each FEBE combination: <FEBE name>-DATAPAR.fits

The naming convention for one MBFITS directory is: DATE-UT-PROJID-OBJECT. The GROUPING.fits file stores the locations of the member files so that the dataset can be accessed as one entity. The GROUPING.fits file primary header also contains some relevant keywords for the Archive, like, e.g., OBJECT, EXPTIME and PROJID, while the Primary headers of the other files in an MBFITS contain only the small set of keywords needed to define the file type as FITS and are thus not useful for the archive. The secondary HDU of each fits file composing an MBFITS contains some relevant keywords related to the content of that file (e.g., HDU 1 of the *DATAPAR.fits file contains the FEBE keyword), and the keywords needed to describe the data table contained in that file. The complex structure of the MBFITS format causes a redundancy of keywords among the GROUPING.FITS file primary header and the HDU 1 headers of the other files, thus some keywords are found in more than one header.

From the Archiving system point of view, the most relevant step is to exactly locate the information (file path, file name, HDU number and exact name of the keywords) to be extracted and organized in the database. To do so, a one-to-one relation is set between non–multiple parameters with keywords location. For multiple parameters like the frequency for each spectral section, a recursive search on frequency domain is performed.

Table 1. HDU structure of a Medicina single-dish FITS file

Index	Extension	Type
0	Primary	Image
1	SectionTable	Binary
2	RFInputs	Binary
3	FeedTable	Binary
4	DataTable	Binary
5	AntennaTempTable	Binary
6	ServoTable	Binary

2.2 FITS

Currently, SD data for the Italian radio telescopes are written in FITS format as a series of binary tables according to the structure described in Table 1. The various HDUs contain information needed to describe the observation and the instrument setup, e.g., in the case of a multifeed receiver the FEED_TABLE describes the feeds, their relative position and power. Each observation of a given source performed with a given instrumental setup (a scan) may be composed of one or more FITS files (one for each subscan). All the FITS files belonging to the same scan are written inside a common folder (hereafter scan folder) together with a Summary file in standard FITS format whose primary header contains all the relevant metadata for the archive. A scan may thus be seen as the "atomic unit" for single dish observations. Single-dish FITS scan folder names are composed as: DATE-UT-PROJID-OBJECT, while the single FITS file naming convention is: DATE-UT-PROJID-OBJECT-SCAN-SUBSCAN. A naming convention to univocally associate the Summary file with the correspondent scan has been established as well. In some cases, typically for spectroscopic observations, the instrumental setup may change during the scan execution (e.g., due to earth rotation the frequency of a spectral line may shift during the observation, thus the observed band must be shifted accordingly to compensate and keep the line at the band center). It has been agreed that in any case the Summary.fits file will always contain only the setup of the first subscan.

2.3 VLBI XML customized format

Each VLBI observation is executed by a variable number of antennas and described by means of proper schedules (VEX files shared between observation sites). Signal correlation is performed offline by using the information in the VEX and log files as well as the observed data themselves. The correlation process generates a Visibility file containing UV data, structured as a monolithic FITS file of some Gby in size. This file contains the real raw-data for an interferometric experiment and is given as input to the data reduction tools for calibration, filtering and image reconstruction. Data inside a Visibility File are homogeneous and contain all the calibration information essential for subsequent data processing. Contrary to what happens for a Single Dish scan, the Visibility File contains all the data coming from an observing session and thus may include data from more than one source. The Visibility File is to be considered as an "atomic unit" and as such will be archived and delivered to the users. The header of the Visibility file contains some keywords needed by the most commonly used packages for data reduction like AIPS, CASA and MIRIAD. However the keyword set is by no means complete in terms of the most commonly used query parameters (for instance, RA and Dec information are not written explicitly in keywords), and other relevant search parameters like field size or resolution are set only after the data reduction process is executed. Also, the FITS file structure may vary in number and position of the HDU depending on the type of observations (imaging, satellite tracking, pulsar observations, etc.). At the same time, the VEX file associated to the observation contains all the information that is relevant for the archive search, but given its structure it is not easy to be parsed in order to fill the database tables with all the relevant metadata. For these reasons it has been agreed that each FITS Visibility Data file is accompanied with a Summary file (similar to the one used for single-dish FITS scans) written in XML format, which lists all the relevant information organized in blocks of keynames/keyvalues, one block for each source contained in the FITS Visibility Data file.

3. ARCHIVING STRATEGY

In order to handle commonly all the three types of radio data formats coming into the Archiving system, a specific architecture and a configurable software has been foreseen.[3] The main characteristics of the archiving system software are the storage of data models into a dedicated database (called data_model schema in a MySQL DBMS) and the configuration of the software behavior using specific information for each instrument (or telescope) saved in dedicated tables of the data_model. The software, called Nadir EXTension for Radio (NEXTR), is build on top of TANGO* control framework and de facto is a TANGO[4] device called RadioDataImporter (RDI). All the metadata relevant for query purposes are extracted from the various set of files and stored into a relational database called radio_metadata. The radio_metadata schema maps the entire MBFITS metadata datamodel plus some administrative dedicated information.

3.1 RadioDataImporter (the software)

Radio Data Importer (RDI) is a TANGO server for importing files (containing observation metadata) in the Radio archive database, currently implemented for MySQL 2 on the Linux operating system. RDI can be configured to import (a subset of) data from either FITS, MBFITS or XML input files. The RDIs configuration is stored in a datamodel database, described in next sections. This configuration enables to store the radio data at various locations, each consisting of a database and a storage directory. More precisely, for each observation instrument that produced an input file, a different destination and subset of data can be selected. All the files that are part of one VLBI-IT observation or of one Single-Dish scan observation must be packed in a single tar or tar.gz archive to be given as input to RDI.

3.1.1 Input File Processing

RDI is programmed to detect and ingest all files of known format that are present in a specifc directory. RDI starts processing the input files when the On command is executed. At the beginning, it reads all the existing files in the input directory and, in addition, it activates the notifications on newly created input files, sent by the Linux Inotify 5 service. A special routine to check the end of writing of new files is foreseen in order to avoid that large files or net latency problems may induce RDI to start managing files before they are fully written. Next step is to unpackage (untar) the previously defined observational "atomic units" and to start the identification of the instrument/telescope type. This is done by reading the appropriate FITS/MBFITS/XML keyword. If the keyword cannot be found, the file is assigned to a default instrument usually defined as "warning" and RDI does not store any data from the source files but just a row with the basic information of the input file (such as the name and the storage directory). The appropriate RDI property enables to select the instruments whose input files are to be processed. If the input file was produced by a unidentified instrument, it is also assigned to the default instrument. Then, RDI reads the data from the source files and archives in the database all the keywords that are included in the data model. Some source files are grouped together since they logically contain the same type of data, but differ in data values. For example, an MBFITS input file can contain several ARRAYDATA files that are in the same group. The extracted metadata are inserted in a set of tables in the Radio archive database. During the insertion, the relations between the rows in related tables (i.e., the foreign keys) have to be defined by taking into account the identifiers. An identifier is a ⟨name, value⟩ pair assigned to a row in the database table, where one row can have multiple identifiers. Identifiers are not inserted in the database (unless explicitly defined in the datamodel database), thus they are neither primary nor foreign keys. An example of an identifier is the FEBE name as part of the ARRAYDATA file name. The definition of (groups of) source files, the database tables, the table columns, and the identifiers are stored in the datamodel database, and read at the RDI initialization. RDI reads the source files by iterating over the defined database tables. For each table, all the source files are checked in order to determine whether they contain data and/or identifiers for this table. This is done by testing whether the source file name matches the information stored in the datamodel database (thus the source files are not opened at this step). When the data from a source file is related to a database table, the source file is opened and data and identifiers are read.

*http://www.tango-controls.org/

3.1.2 Keywords and Identifiers

Data and identifiers are defined by keywords and file headers that contain these keywords. When reading the value of a keyword, several cases may occur with respect to the definition of the keyword itself:

1. A keyword can store several values (that are organized in multidimensional arrays). In this case, several values are returned.
2. A keyword can be defined with a pattern. If, in addition, it is defined as countable, then the pattern is replaced with a natural number, starting with 1, and the keyword is read. Afterwards, the number is increased and the keyword with the increased number is read. This increase-number-and-read-keyword procedure continues until such a keyword exists. On the other hand, if the keyword is not defined as countable, then all the keywords are checked to verify if they match the pattern. For each keyword matching the pattern, or the pattern plus natural number, a single value is read. The values of all the matching patterns are returned.
3. A keyword can store the number of rows. In this case, a single value is read. Afterwards, a set of natural numbers is defined storing numbers from 1 to the read value. Finally, this set is returned.
4. A keyword can store a single value. In this case, only one value is returned.

When combining the values of the keywords related to a database table, the following cases can occur:

1. The size of the values of a keyword is the same as the size of values of other keywords. In this case, values are combined together to form several rows, i.e., the first values of all keywords are combined into the first row, the second values into the second row etc.
2. If the size of the values of a keyword is equal to one, then this keyword value is added to all the rows.
3. If the size of the values of a keyword is not the same as the size of values of other keywords, and is not equal to one, an error is produced (assigning the input file to the "warning" instrument).
4. An error is also produced if the value of a mandatory keyword is missing.

The same cases also occur when reading keywords of table identifiers. However, there are few differences, namely: the identifier values are not added to rows as data, but as metadata; the number of rows can be a multiplier of the number of identifier values. Consequently, when assigning identifier values to rows, the set of identifier values is repeated until all the rows receive an identifier value. Moreover, there exists three (plus default) types of identifiers:

1. Content identifier: its value is read in the same way as the table data values, and assigned to the rows as described above.
2. Row index identifier: it is used to assign a row/column index to the values of multidimensional keywords. Currently, one- and two-dimensional data are supported. For each dimension, the number of elements in the array must be given. The sizes of two-dimensional data are written in a table, meaning that for each row/column a custom size can be defined. In addition, the first identifier represents the number of rows, while the second is the number of columns. When the size(s) of the dimension(s) are determined, a set of one/two dimensional indexes is created. For example, the rows of one-dimensional data get the indexes (0, 1, 2, 3, ...), while the rows of two-dimensional data get the pairs of indexes (<0,0>, <0,1>, ... <1,0>, <1,1>, ...).
3. File_ name identifier: its value is obtained by parsing the name or path of the source file. This value is added to all the rows.

4. Default identifier: the previously described identifiers have to be defined in the database. However, if no identifier is defined for a database table and if there is only one file containing data for this database table, the default identifier is applied. This identifier assigns sequential numbers to the rows of data. While the previously described identifiers are mostly used for the FITS and MBFITS files, the default identifier is mostly used for the XML files that contain no obvious identifier.

A database table may store data from several groups of source files, where each group contains data for a subset of table columns. Note that there must be no intersection between column subsets of various groups of source files. In this case, data from various groups are combined in rows using the following procedure. After the first group of source files related to a table is processed, no new rows can be added to the table. When processing other groups of source files related to the same table, the obtained rows must be merged with the existing ones. More precisely, for each new row, a related existing row must be found by matching the identifiers (see the description of the matching procedure below). When the related row is found, these two rows are concatenated together meaning that all the data from the new row are simply added to the existing row. This enables to merge data from two or more (groups of) source files related to the same table, where each of them store data for its own subset of table columns. When all the data and identifiers of all the tables are read, the relations between all the rows of all the related tables are found (i.e., the foreign and primary keys of the related rows are connected together). This is done using identifiers as follows. When table A is related to table B and table B requires the foreign key from table A, then for each row in table B a row in table A that matches the identifiers is found. The matching can be partial, meaning that it is not required that both rows have the same set of identifiers. Nevertheless, it is required that if both rows have identifiers with the same name, their values have to be the same. In addition, it is also mandatory that each row in table B matches to one and only one row in table A. If any of these requirements is not fulfilled, the input file processing is not successful and the input file is assigned to the default ("warning") instrument. Finally, there exists a table that stores only one row for each input file (containing some general information). This is the only table that does not store foreign keys from other tables storing data (i.e., the "data_file" table in Figure 4 and Figure 5). If the number of rows to be inserted in this table is not exactly one, the input file processing is not successful. However, this condition is checked only for the FITS and MBFITS files, since an XML file can have multiple entries in the "data_file" table. Note that RDI uses two libraries to read the source files: one for FITS/MBFITS, the other for XML. When a file has to be opened and read, the appropriate library is selected based on the extension of that file.

3.2 Data insertion in the Radio archive database

The insertion of data in the Radio archive database is done as follows. First the input file basic data, such as the file name and the storage path, are inserted in the main table (i.e., the file_info table in Figure 4 and Figure 5), which also stores the input file version. The file version is obtained by checking all the existing files in the database with the same name and determining their maximum file version. If necessary, the maximum file version is increased by one and the new file version is assigned to the input file. This procedure has to be thread-safe since multiple input files with the same name can be simultaneously inserted in the same database (from one or more instances of RDI). This is achieved by first locking the main table, then reading the maximum file version and increasing it, storing the basic information of the input file (including the file version) in the main table, and finally unlocking the main table. Such a procedure also enables other threads to read/lock the main table without waiting for the current thread to store all the other data in the database. When the basic input file data are stored in the main table, all the other data can be inserted in the database. This is done by starting a transaction, storing the data in the database tables, executing the procedures that are defined in the RDI configuration, copying the input file in the storage directory, and stopping the transaction. Note that RDI only stores the data as they are found in the source files. If some processing is needed, data must first be inserted in the database tables and then already defined procedures have to be applied. If the input file was successfully copied and data were successfully inserted in the database, the transaction is stopped with a commit and the input file in the input directory is deleted. Otherwise, the transaction is stopped with a rollback, the basic input file data are deleted from the main table (since this insertion is not in the transaction), and the input file is not deleted.

3.3 Datamodel database

The datamodel database stores the configuration of RDI regarding the set and the structure of data that has to be read from the source files and inserted in the Radio archive database. To this aim, it stores the data related to the various instruments, the information on the location of the Radio archive databases and storage folders, the structure of the Radio archive databases (names of tables, relations among them, columns of each table), and the description of the source files obtained when untaring the input file (their names and the data that can be found in each file). The following Sections describe the groups of tables of the datamodel database.

3.3.1 Instrument definition

The datamodel database defines one or more destinations, each defining an instance of the Radio archive database (i.e., it stores connection data including schema, name and type, e.g., MySQL, Oracle, etc.) and a storage directory. One destination can store data from various instruments, each uniquely defined by its name. In addition, a destination is associated with a directory name where the input files are stored, and with the name of the main table in the Radio archive database (file_info_table_name). Moreover, via an internally-defined type name, it stores the information on the type of data contained in the input file. For example, if the input file stores an MBFITS, the data type can be fits-mb. When an input file is inserted in the Radio archive database its data type is copied in the FILE_INFO table. This enables to distinguish among input files based on their type. There can be several configurations defining the data to be read from the source files and stored in the Radio archive database. Each one (except for the default, warning instrument) has an associated instrument_configuration used to read data from each input file coming from that instrument. The instrument configuration defines the keyword storing the instrument name (foreign key data_id_instrument_name) and the keyword storing the observation date (foreign key data_id_date_obs).

3.3.2 Configuration definition

A configuration defines the tables and the structure of the Radio archive database. In addition, it defines the data that has to be inserted in these tables, and the source files where these data can be found.

Definition of tables

Tables storing the data from the source files are listed in the database_table so that no previous table needs a foreign key from any subsequent one. Besides the table name, the result_id is stored as the name of the primary key column. When a row is inserted in the Radio archive database, the value of the primary key is automatically created by the autoincremental function and associated to the result_id. This value is then used as a foreign key for the connected rows in the connected tables. If result_id is *null* then the table has no autoincremental primary key. The relations between Radio archive database tables are stored in precedent table (parent reference). When inserting a row, the precedents enable to determine the tables to which the row has to be connected. The other_data defines the other data that has to be inserted in each row of a Radio archive database_ table. Currently, it is used to determine which foreign keys from connected tables have to be inserted in each row. The table_procedure stores the names of the procedures that have to be executed after all the data of an input file are inserted in the Radio archive database.

Definition of data of source files

The source_file table stores the path and name patterns that are used to search for the source files of the same type, i.e., which contain the same data structure but differ only in data values. Therefore, a datum can be stored in several files where, in most cases, one row is created for each file. However, if the datum stores a table or if multiple instances of the keyword are stored in one file (see the description of data below), then multiple rows are inserted in the Radio archive database for each file. If multiple files store data in the same table, then two cases can be distinguished:

1. all the files contain data that can be stored in one row. In this case, these data are simply concatenated in one row.

2. some files contain data that have to be stored in multiple rows. For example, the GROUPING file stores multiple FEBEn, while FEBEPAR files store data related to specific FEBEs. Both data could be stored in the same table (related to FEBE data). In this case, the concatenation of rows of these files has to be determined using identifiers (see the description of identifiers below). In this example, the FEBE identifier has to be determined for data from GROUPING and for data from FEBEPAR. Then data with the same FEBE value have to be concatenated in the same row.

Data definition

The table data stores the definition of each datum that has to be read from the source file and inserted in the Radio archive database. This definition consists of:

- the file where the datum is stored (i.e., the foreign key to the source_file);
- the part of the file where it can be found, such as header1, data0; the current implementation supports only "header"+integer and "data"+integer syntax for the purpose of processing the FITS and MBFITS file, while when processing the XML files, the file part is ignored;
- keyword pattern *pri* and *sec*: first the primary keyword has to be checked and, if it does not exist, the secondary keyword is checked;
- the column name where the datum will be inserted in the Radio archive database;
- type [char, double, int, bool, date]; consistency with the type in the source files is checked and the input file is assigned to the default instrument if a datum type is not consistent;
- countable: used only when a keyword represents a pattern, i.e., when it contains " .* "; if it is countable, then .* is replaced with a sequence of natural numbers until such a keyword is not found (e.g., FEBE1, FEBE2, FEBE3, FEBE4, ... is read until no more found); if not countable, the list of keyword are checked one by one to verify if it match the pattern;
- are_multiple_dimensions: used only when file HDU part contains "data" to be read; it represents the number of dimensions of a datum, i.e., 0 = single data, 1 = 1-D array, 2 = 2-D table etc;
- is_number_of_rows: if true then the datum represents the number of rows that have to be inserted in database; in this case, a row is inserted for each natural number from 1 to the value of the datum. For instance, CHANNELS from ARRAYDATA store the number of rows that could be inserted in a table related to channels.;
- mandatory: if true and this datum does not exist in the source file, then the input file is assigned to the default instrument ("warning");
- read_as_single_datum: if true then all the bits storing the value of this datum are read as a single value; e.g., a file might contain 3 values for a datum, each of them stored in 24 bits, if read_as_single_datum is true, then only one datum of 3*24 bits size is read if multiple instances of one keyword are feasible; as an example, FEBEn in GROUPING, would be defined as a pattern, e.g., FEBE.*

Data that are stored in each table are defined in step_data.

Definition of data identifiers

To determine the relations, i.e., foreign keys between the rows of data inserted in connected tables, one or more identifiers have to be defined. Each identifier has a name that enables to compare identification values of different rows. Therefore, when two rows are compared in order to determine if they are connected, the values of the identifiers with the same name are compared. If the values are equal, these rows are connected, i.e., they share foreign keys. For example, a data from the FEBEPAR file has an identifier called FEBE and its value is a part of the name of the file. A data from the ARRAYDATA file has another identifier with the same name (FEBE)

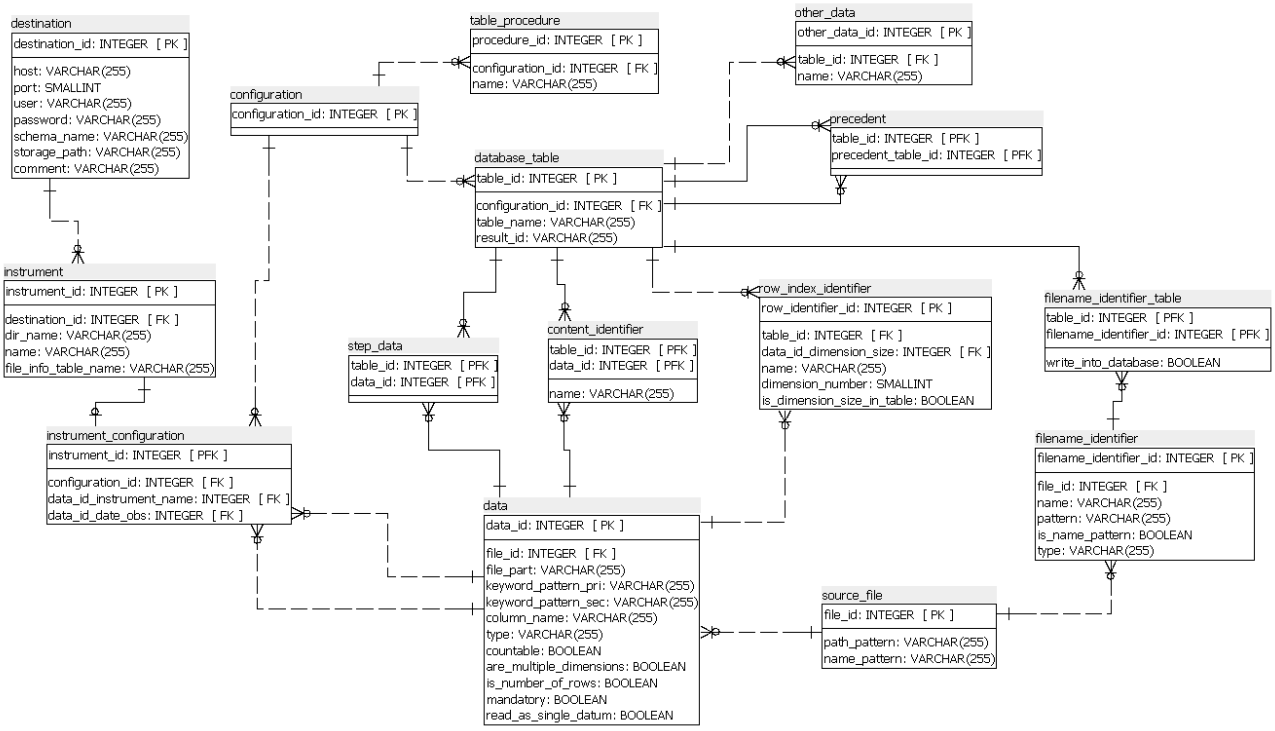

Figure 1. Data model schema tables

and its value is also read from the part of the name of the file. These two identifiers with the same name enable to (transitionally) connect the data from FEBE and ARRAYDATA, i.e., the data with the same FEBE value will be (transitionally) connected with foreign keys. The following types of identifiers exist, depending on the source of the identifier value:

- content_identifier: this is the most common identifier. Its value is obtained by reading a datum from the source file. For example, FEBEn from GROUPING represents a content identifier, since its value is the identifier for the rows related to FEBE;
- filename_identifier and filename_identifier_table: a file name and its path may contain an identifier. In this case, one or more patterns are defined, which enable to extract one or more filename identifiers. A filename identifier has a pattern that defines where the identifier value can be found (the position is denoted with parentheses). For example, " .*-ARRAYDATA-([0-9]*)..* " is a pattern to obtain the baseband number from the ARRAYDATA file name. Next, the pattern type is defined (int, char). If is_name_pattern is true, this is a file name pattern, otherwise it is a file path pattern. If write_into_database is true, this identifier has to be written into the database for each row that it identifies. An example of filename identifier is the file DATAPAR that contains the subscan in its path, and the FEBE in its name (two filename identifiers);
- row_index_identifier: the source file might also store data that are tables of values, where the row index represents the identifier. For example, FEEDOFFX from FEBEPAR stores a value for each FEED of a receiver, where the row index is equal to the FEED index. If the datum represents multidimensional data, then one row_index_identifier is defined for only one dimension of these data. Consequently, multiple row_index_identifiers can be defined on the same multidimensional data, each index for a different dimension. The following data have to be defined for each row_index_identifier:
  - dimension_number: row identifiers on one table (one datum) must have unique dimension numbers that are sequential natural numbers, i.e., 1, 2, 3, 4, etc. Therefore, 1-D table can have one identifier

with dimension number 1; 2-D table two identifiers with number 1 and 2; 3-D table three identifiers with number 1, 2, 3 and so on;

- data_id_dimension_size: this is the foreign key to the datum that stores the number of elements in row/column where the identifier is defined; if is_dimension _size_in_table is true, then it stores multiple values, one dimension size for each row/column;
- is_dimension_size_in_table: true if the size of one dimension is stored in table; e.g., if the dimension represents rows, the table stores the number of rows for the first, second, third etc. column; this enables to have different number of rows for each column.

There exists also a default identifier. Note that this identifier is not inserted/stored in the datamodel database.

3.4 General MBFITS database

A general MBFITS database is used as a baseline for the creation of the Radio archive database. The Radio

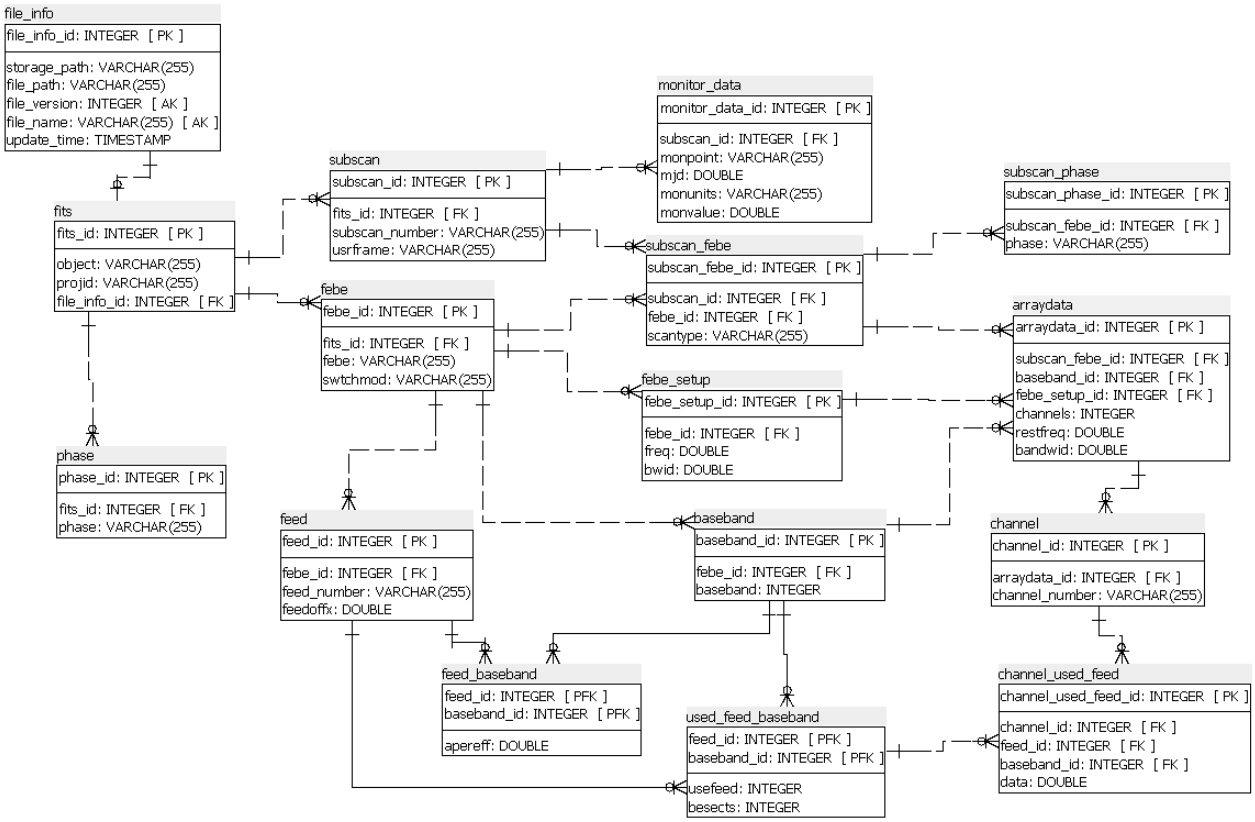

Figure 2. MBFITS database including all the needed tables

archive database (as described in the following sections) is then used to store data from the source files on the basis of the configuration stored in the datamodel database. The configuration enables to select from the source files the subset of data that has to be stored in the Radio archive database. To do this, the Radio archive database itself consists of a subset of MBFITS database tables containing subsets of MBFITS database data. In any case, the structure of the Radio archive database tables must comply with the data structure in the MBFITS database and in the FITS/MBFITS/XML files. This section presents all the possible MBFITS database tables that can be defined based on the definition of the MBFITS file format. Therefore, all the data in any MBFITS can be stored in the presented database structure. To maintain the description compact, only a subset of data will be shown in each table. Nevertheless, all the tables, thus the entire database structure will be presented.

Note that the data in the FITS and XML files represent only a subset of data in the MBFITS files, thus the data from the FITS and XML files can be easily mapped/stored in the MBFITS database. Figure 2 shows the detailed database structure for a generic MBFITS data model rapresentation (full model).

3.4.1 Description of tables and data cardinality

Table 2 shows the symbols that are used to describe the MBFITS database. The tables used to store the MBFITS data are described in Table 3. The MBFITS database including all the needed tables and a subset of data columns is shown in Figure 2.

Table 2. Symbols describing the MBFITS database

Symbol	Description	Value	Index
N_{FEBE}	Number of FEBEs	<i_{FEBE}>-FEBEPAR (= SCAN.NFEBE)	i_{FEBE}
N_{SCAN}	Number of SubScans	SCAN.NSUBS	i_{SCAN}
$N^{i_{FEBE}}_{BAND}$	Number of used baseband for FEBE	<i_{FEBE}>-FEBEPAR. NUSEBAND	i_{BAND}
$N^{i_{FEBE}}_{FEED}$	Number of FEEDs for FEBE	<i_{FEBE}>-FEBEPAR. FEBEFEED	i_{FEED}
$N^{i_{FEBE},i_{BAND}}_{USEFEED}$	Number of used FEEDs for baseband for FEBE	<i_{FEBE}>-FEBEPAR.NUSEFEED [i_{BAND}] repeated in <i_{FEBE}>-ARRAYDATA- <i_{BAND}> NUSEFEED	$i_{USEFEED}$
$N^{i_{SCAN},i_{FEBE},i_{BAND}}_{CHANNEL}$	Number of channels for a baseband for FEBE during a subscan	i_{SCAN} /<i_{FEBE}>-ARRAYDATA- <i_{BAND}> CHANNELS	$i_{CHANNEL}$
$N^{i_{SCAN}}_{MONITOR}$	Number of monitor raw data for a subscan	MONITOR.BINARY_TABLE	$i_{MONITOR}$
N_{PHASE}	Number of phases	SCAN.PHASEn	i_{PHASE}
$N^{i_{FEBE}}_{PHASE}$	Number of phases per FEBE	<i_{FEBE}>-FEBEPAR.NPHASEs	i_{PHASE}

Table 3. MBFITS database tables and data cardinality - 1

Table Name	Number of Entries per MBFITS	Source of Data	Description
FILE_INFO	1		Only single-value data
DATA_FILE	1	GROUPING SCAN	Only single-value data
FEBE	N_{FEBE}	$<i_{FEBE}>$-FEBEPAR	
SUBSCAN	N_{SCAN}	i_{SCAN}/MONITOR	Each subscan corresponds to one subfolder, where the subfolder name represents the SUBSCAN name
SUBSCAN_FEBE	N_{SCAN} N_{FEBE}	i_{SCAN} /$<i_{FEBE}>$ -DATAPAR	One row for each FEBE in each subscan. It also stores values that are different than the default values stored in SCAN, e.g., SCANMODE
BASEBAND	$\sum_{i_{FEBE}=1}^{N_{FEBE}} N_{BAND}^{i_{FEBE}}$	$<i_{FEBE}>$-FEBEPAR binary table only	Only data related to basebands, e.g., BOLCALFC. Names are stored in USEBAND, however, they are not ineeded since only baseband indexes are used
ARRAYDATA	$N_{SCAN} \sum_{i_{FEBE}=1}^{N_{FEBE}} N_{BAND}^{i_{FEBE}}$	i_{SCAN} /$<i_{FEBE}>$ -ARRAYDATA- $i_{BAND}>$	One row for each baseband of each FEBE for each subscan
FEED	$\sum_{i_{FEBE}=1}^{N_{FEBE}} N_{FEED}^{i_{FEBE}}$	$<i_{FEBE}>$-FEBEPAR binary table only	Only data related to feeds, e.g., FEEDOFFX

3.5 Radio Archive Database

The Radio archive database is a subset of the general MBFITS database described in Section 3.4. Data that are required to be stored in this database are described in Figure 3. Note that the presented database is a common

Table 4. MBFITS database tables and data cardinality - 2

BASEBAND_FEED	$\sum_{i_{FEBE}=1}^{N_{FEBE}} (N_{FEED}^{i_{FEBE}} N_{BAND}^{i_{FEBE}})$	<i_{FEBE}>-FEBEPAR binary table only	Only data 2D tables related to basebands and all feeds e.g., APEREFF
BASEBAND_USED_FEED	$\sum_{i_{FEBE}=1}^{N_{FEBE}} (\sum_{i_{BAND}=1}^{N_{BAND}^{i_{FEBE}}} N_{USEDFEED}^{i_{FEBE},i_{BAND}})$	<i_{FEBE}>-FEBEPAR binary table only	Only data 2D tables related to basebands and used feeds e.g., BESECTS. Names of used feeds are stored in USEFEED
CHANNEL	$\sum_{i_{SCAN}=1}^{N_{SCAN}} (\sum_{i_{FEBE}=1}^{N_{FEBE}} (\sum_{i_{BAND}=1}^{N_{BAND}^{i_{FEBE}}} N_{CHANNEL}^{i_{SCAN},i_{FEBE},i_{BAND}}))$	i_{SCAN} /<i_{FEBE}>-ARRAYDATA-<i_{BAND}>, only CHANNELS keyword	For an ARRAYDATA, the number of created rows is equal to CHANNELS. Each row stores a number from 1 to CHANNELS
USED_FEED_CHANNEL	$\sum_{i_{SCAN}=1}^{N_{SCAN}} (\sum_{i_{FEBE}=1}^{N_{FEBE}} (\sum_{i_{BAND}=1}^{N_{BAND}^{i_{FEBE}}} N_{CHANNEL}^{i_{SCAN},i_{FEBE},i_{BAND}} N_{USEFEED}^{i_{FEBE},i_{BAND}}))$	i_{SCAN} /<i_{FEBE}>-ARRAYDATA-<i_{BAND}>, only DATA from binary table	The FEED order is the same as in <FEBE>-FEBEPAR. USEDFEED
MONITOR_DATA	$\sum_{i_{SCAN}=1}^{N_{SCAN}} (N_{MONITOR}^{i_{SCAN}})$	i_{SCAN} /MONITOR binary table only	Binary table of MONITOR stores raw monitoring data, one row for each data
PHASE	N_{PHASE}	SCAN, only PHASEn	For each PHASEn, one row is created.
SUBSCAN_PHASE	$N_{SCAN} \sum_{i_{FEBE}=1}^{N_{FEBE}} N_{PHASE}^{i_{FEBE}}$	i_{SCAN} /<i_{FEBE}>-DATAPAR, only PHASEn	For each PHASEn, one row is created. Phases are stored in DATAPAR only when they are different than phases stored in SCAN

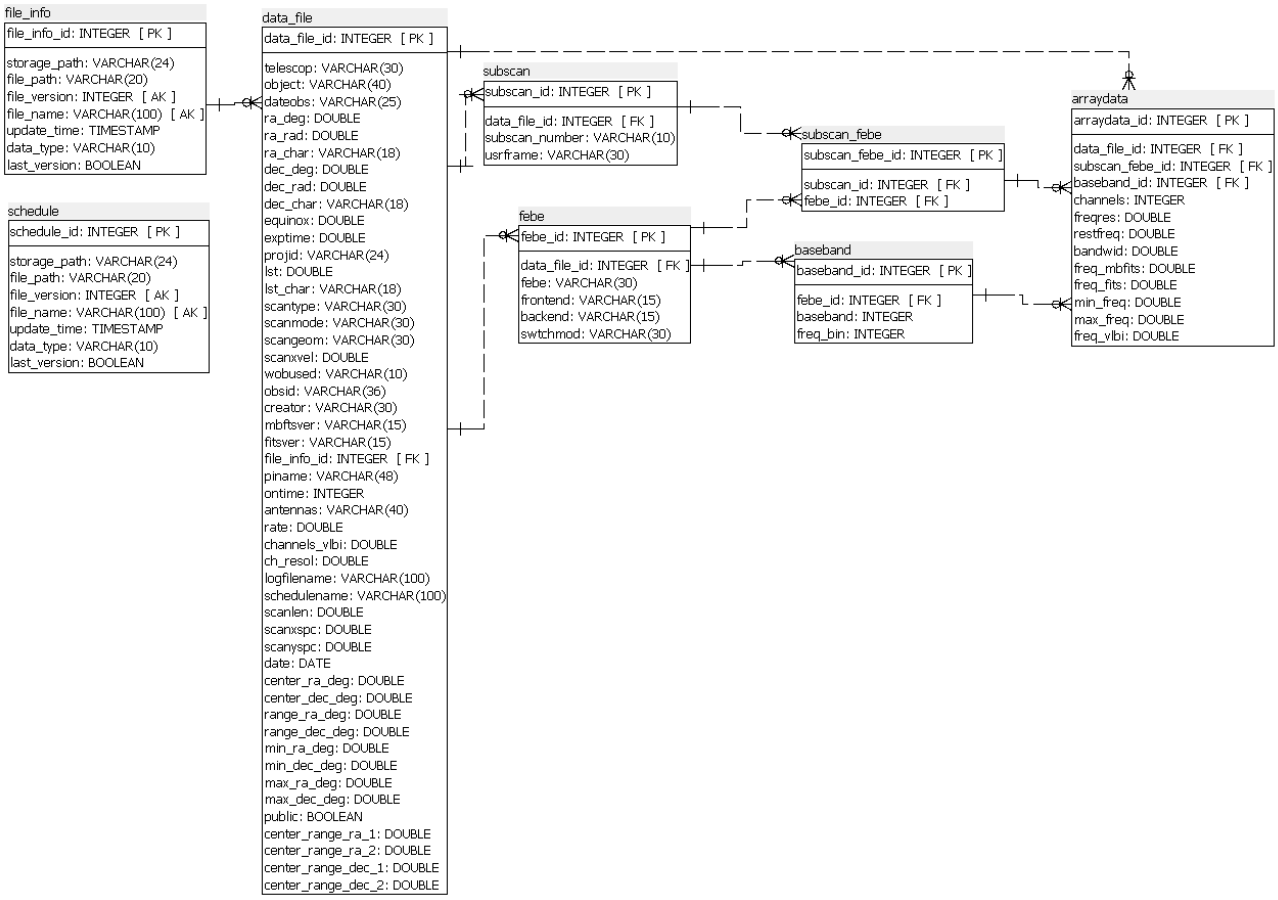

Figure 3. Radio Archive database including only the needed tables

database for the FITS, MBFITS and XML files. Therefore, the data_file table contains fitsvers, mbfitsvers and freq_vlbi, where only one of them is filled depending on the type of input data. All the three data types can be stored in the database. A few metadata have different meaning depending on the type of input data. For instance, the keyword FREQ describing the frequency can represent either the minimum or the average frequency in the observing band. The database is able to recognize and handle both meanings of FREQ thanks to a stored procedure used to calculate the minimum and the maximum frequency and store them in min_freq and max_freq. If some input files are not valid, data cannot be read and a row is inserted only in the main table called file_info. This row has no associated entry in the data_file table. This enables to recognize problematic input files. The Radio Archive database contains an additional table (called schedule table) with respect to the MBFITS database. The schedule table is a copy of file_info table, which is not linked to the tables storing data from the input files. Such a configuration enables to store two input files for each observation, i.e., an input file storing the actual observing data, and an input file storing the log and the schedule of the observation. To handle both types of input files, the deployment of two instances of RDI is required. The first instance reads the data from FITS/MBFITS/XML and stores them in file_info, data_file and other connected tables. The second instance processes the schedule input files without untaring/opening them, but just moving them in the storage folder and inserting their names in the schedule table. Note that these two instances of RDI read the input files from two different input folders, each of them storing one type of input files. When searching for the schedule of an observation, the schedule name in the data_file table has to be matched with the file_name in the schedule table.

3.6 User access

Users can perform dedicated queries in the Radio Archive depending on Instrument (Telescope) features or modes. In particular, SD and VLBI-IT observation share a generic subset of parameters and can be specialized for each instrument in terms of selection parameters and query outputs. Common features are, for example, the user authentication page, the celestial object name resolver, the coordinates and the frequencies fields. Specialized fields are related to the Observing mode, the frontend/backend configuration and so on. To improve the query performance and speed of results delivery, an indexing process of the most commonly used columns and SQL functions were implemented. The index process was set up using the indications of a group of test users (professional astronomers) while the querying functions have been implemented to optimize the investigation over the hierarchical database structure, returning information organized in a table on the output web page. To speed-up the queries, two approaches were applied. First, all the parameter computations that could be done during the query execution are performed in advance and stored in the database. In this way, there is no need for real time calculation and the computed parameters are easily indexable. Second, indexes were created on various columns and combinations of columns to speed-up the most frequent queries. These processes are very specific and customized on the necessities of a Radio Archive query form, and on the effective necessities of the average radio astronomer curiosity and needs. Moreover, a dedicated algorithm was implemented for the search inside a circular region around user-selected celestial coordinates on rectangular radio maps with different reference frames.

3.7 Conclusions

In view of the realization of a public Radio Data archive, the design and development of a comprehensive data model for radio observations taken with the Italian radio telescopes required a thorough analysis of the available instrumentation. Beside this, a configuration mapping for software set up has been elaborated in order to delegate all the data model complexity to a defined number of possible schemes designed to be human readable and easy-to-use as far as possible, to allow software re-usability and flexibility. By changing the Radio_data model using simple SQL commands, different kinds of instruments/telescopes can be easily added to the Radio Data Archive. Modifications to the source data models are also easily handled, thus offering the possibility to have different versions of the Radio_data model working at the same time. The choice to be fully compliant with the MBFITS data model improved the perspective of the Archive in the direction of VO compliance. Moreover, a Raw Radio Data model which is already in use by the astronomical community may be of interest for the Virtual Observatory Alliance to define a new model for this kind of data, or to include this specific model into an already existing one, opening the possibility to use Table Access Protocol (TAP) to export publicly available resources. An ongoing activity in this field is foreseen to export the Radio Archive Database to VO compliant clients through TAP, in view of improving open data re-usage and the accessibility of scientific data to the general public.

3.8 Acknowledgments

This work became feasible due to the great contribution of both the Radio Astronomy Institute of Bologna and Medicina and the Italian Astronomical Archives data center teams.

REFERENCES

[1] Murder, D., "Multi-beam fits raw data format," *APEX Interface Control Document* **1**, – (2015). http://www.apex-telescope.org/documents/public/APEX-MPI-ICD-0002.pdf.

[2] Zanichelli, A., "Data formats for the medicina and noto radio telescopes," *IRA Technical Report* **1**, – (2015). http://www.ira.inaf.it/Library/rapp-int/488-15.pdf.

[3] Dovgan, E., "Radio data importer report," *OATs Technical Report* **1**, – (2015).

[4] Chaize, J.-M., "Tango- an object oriented control system based on corba," in [*7th International Conference on Accelerator and Large Experimental Physics Control Systems*], D. Bulfone, A. D., ed., *Proc. ICALEPCS* **1898**, – (1999).