
	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 1 of 40
---	----------------------------	--	--

Radio Data Importer Report

Document No.: IA2-ARC-035 / OATS Technical Report n. 206
Issue/Rev. No.: 1.0
Date: 16 February 2016
Prepared by: Erik Dovgan, Cristina Knapic, Riccardo Smareglia
Approved by: Riccardo Smareglia

INAF – Osservatorio Astronomico di Trieste, via G. B. Tiepolo 11, I-34131 Trieste, Italy

This document contains colour images, which can be printed in monochrome.

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 2 of 40
---	----------------------------	--	--

Document Approval

Name	Organization	Signature	Date
Riccardo Smareglia	INAF-OATs	PDF Digital Signature	16 February 2016

Document Status Sheet

1. Document Title: Radio Data Importer Report			
2. Issue	3. Revision	4. Date	5. Reason for change
0.2	1	10/06/2015	Added description of XML handling
0.3	1	18/06/2015	Added changes after meeting with IRA
0.4	1	18/08/2015	Added startup script, updated VLBI config, added validation table
0.5	1	24/09/2015	Added database querying
0.6	1	25/11/2015	Added database replication
1.0	1	16/02/2016	Finalization of the report




	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 3 of 40
---	----------------------------	--	--

Table of Contents

1.	Introduction	5
1.1	Acronyms.....	5
1.2	Reference documents.....	5
2.	Radio Data Importer.....	6
2.1	States and commands	6
2.1.1	The Init command.....	6
2.1.2	The On command.....	6
2.1.3	The Off command.....	6
2.1.4	The state of reusable connections	7
2.2	Input file processing	7
2.3	Reading data from source files	8
2.4	Data insertion in destination Radio archive database.....	11
2.5	Linked libraries and dependencies.....	11
2.6	Installation and configuration	13
2.6.1	Registration in TANGO	13
2.6.2	Device properties.....	13
2.6.3	Automatic startup configuration.....	14
2.6.4	Custom logging.....	14
2.6.5	Applying data access policy	15
3.	Datamodel database.....	18
3.1	Instrument definition	18
3.2	Configuration definition	18
3.2.1	Definition of tables	19
3.2.2	Definition of data of source files	19
3.2.3	Data definition.....	19
3.2.4	Definition of data identifiers.....	20
3.2.5	Configuration use case	21

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 4 of 40
---	----------------------------	--	--

4.	General MBFITS database	24
4.1	Overview.....	24
4.2	Description of tables and data cardinality.....	24
5.	Radio archive database	29
6.	Database querying.....	31
6.1	Data preprocessing and query building.....	31
6.2	Data indexing.....	33
6.3	Query execution.....	35
7.	Database replication	37

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 5 of 40
---	----------------------------	--	--

1. Introduction

This document describes the Radio Data Importer (RDI). RDI is used to import radio (meta)data acquired with the Medicina and Noto radio telescopes into a common Radio archive database. More precisely, RDI processes data stored in the FITS, MBFITS, and custom-defined XML format. RDI can be fully customized by defining configuration data in a datamodel database.


This document describes the RDI data processing procedure, the datamodel database, the database for storing (meta)data from a general MBFITS format, and the database customized for the radio telescopes, i.e., Radio archive database. In addition, it also describes how the replication of the radio (meta)data is achieved.

1.1 Acronyms

RDI	Radio Data Importer
FITS	Flexible Image Transport System
MBFITS	Multi-Beam Flexible Image Transport System
XML	eXtensible Markup Language

1.2 Reference documents

- [1] D. Muders, E. Polehampton, and J. Hatchell; *Multi-Beam FITS Raw Data Format*; Revision 1.63, 5 August 2011
- [2] A. Zanichelli, M. Nanni, S. Righini, A. Orlati, F. Bedosti, M. Stagni, C. Knapic, M. De Marco, and R. Smareglia; *Data formats for the Medicina and Noto radio telescopes in view of a common Archive*; Version 5.5, May 2015

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 6 of 40
---	----------------------------	--	--

2. Radio Data Importer

Radio Data Importer (RDI) is a TANGO¹ server for importing the FITS, MBFITS and XML files (containing observation metadata) in the Radio archive database (currently implemented for MySQL²) on the Linux operating system. It can be configured to import (a subset of) data from either FITS, MBFITS or XML, i.e., input files. The RDI's configuration is stored in a datamodel database, described in Section 3. This configuration enables to store the input file data in various destinations, 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. The input files must be in the “tar” or “tar.gz” format, i.e., all the source files that are part of one observation must be packed in one “tar” or “tar.gz” archive.

2.1 States and commands

2.1.1 The Init command

At the initialization, the state is set to INIT. Afterwards, RDI:

1. Sets the Java library path to native libraries of Inotify for Java³ that is used for detecting the creation of the input files
2. Loads the driver for the database and establishes the connection to the database
3. Reads the configuration from the datamodel database and checks its consistency; for each destination a pool of reusable database connections is created
4. Starts the Executor Services⁴ for processing input files
5. Sets the state to OFF

If configured to start file processing automatically, RDI executes the On command.

2.1.2 The On command

When executing the On command, RDI registers Inotify for Java listener on the selected input directory for the “Create” and “Moved to” (i.e., “Rename”) events. In addition, if the Executor Services are shut down, RDI starts them. If these steps are successful, the state is set to ON, otherwise, the state is set to OFF.

2.1.3 The Off command


The Off command unregisters the Inotify for Java listener and shuts down the Executor Services. Afterwards, the state is set to OFF.

¹ <https://www.tango-controls.org/>

² <https://www.mysql.com/>

³ <https://bitbucket.org/nbargnesi/inotify-java>

⁴ <http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html>

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 7 of 40
---	----------------------------	--	--

2.1.4 The state of reusable connections

RDI maintains a pool of database connections that are reused when needed, i.e., when a new input file is inserted in the database. If the new input files are created rarely, the database connections may be inactive for several time, which can lead to the automatically closure of these connections on the side of database management system. However, RDI is not notified about this closure, which may lead to errors when a (closed) connection is attempted to be reused. To resolve this issue, RDI checks if a connection is still opened before reusing it. If not opened, it is removed from the pool and a new connection is created and opened. To check a connection, a predefined SQL statement is executed, which has to return at least one row. The predefined SQL statements for datamodel database and Radio archive database are presented in Sections 3 and 5.

2.2 Input file processing

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. In addition, it accepts the notifications on newly created input files, sent by the Linux Inotify⁵ service. For each input file, the file writing conclusion procedure firstly checks if the file has the “tar” or “tar.gz” extension. If true, it creates a File Writing Conclusion thread used to check when the file is entirely written. This thread is passed to the Executor Service 1 that is used to maintain a fixed number of active threads for file writing conclusion checking at each time. Other threads wait until an active thread concludes.

When a thread is selected for execution (by the Executor Service 1), it periodically checks the file size and the validity of the tar file format. The period between two checkings is defined as the RDI property. The validity is checked by executing the Linux “tar” command. If it returns some error(s), the tar is not valid. The size and validity are checked until the tar is valid and size does not change, i.e., the file writing has concluded. Then a new thread for processing this input file is created and passed to Executor Service 2.

Executor Service 2 is used to maintain a fixed number of active threads for processing the input files. When a thread is started, it processes the input file in the following steps:

a. Untar the source files from the input file


The source files are untarred from the input file into a temporary folder that is defined as a RDI property

b. Determine the instrument that created the input file

This is done by reading the appropriate FITS/MBFITS/XML keyword. If the keyword cannot be found, the file is assigned to the default (usually defined as “warning”) instrument.

In the case of the default instrument, RDI usually does not store any data from the source files. Nevertheless, it stores just a row that contains the basic information of the input file (such as the name and the storage directory).

⁵ <http://en.wikipedia.org/wiki/Inotify>

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 8 of 40
---	----------------------------	--	--

The appropriate RDI property enables the user to select the instruments whose input files will be processed. If the input file was produced by a non-selected instrument, it is also assigned to the default instrument.

c. Read the data from the source files

Only the data that user selected are read. The selected set of data is stored in the RDI's configuration. The reading procedure is described in Section 2.3.

d. Close and remove the temporary source files

e. Insert data in destination Radio archive database and move the input file to storage directory

This procedure is described in Section 2.4.

Note that if only one, i.e., default instrument is defined, the input file is not untared, the data is not read and the temporary files do not exist and are not removed. This is due to the fact the default instrument should not define data to be read from the source files. Nevertheless, general information about the input file is inserted into the main (file_info) table and the input file is moved in the storage folder.

During the processing of the input file, critical errors can occur, such as no disk space is available or there is a problem with the database. A critical error indicates that the processing of each new input file will also result in a critical error. Therefore, it makes no sense to continue processing input files. Consequently, RDI stops processing the files by executing the Off command. RDI also enables to notify selected system administrator(s) with an email when a critical error occurs. To that end, administrators' email addresses can be stored in the appropriate RDI property.

The interaction among RDI components and between RDI and other components is shown in Figure 1.

2.3 Reading data from source files

The input file can contain one or more source files where the data are stored. 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 [1] that are in the same group. The data from source files have to be inserted in the Radio archive database in a set of tables. 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 that is defined for 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, see Section 3), thus they are not primary or foreign keys. An example of an identifier is the FEBE name as part of the ARRAYDATA file name [1]. The definition of (groups of) source files, the database tables, the table columns, and the identifiers is stored in the datamodel database (see Section 3), 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


	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 9 of 40
---	----------------------------	--	--

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


- 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 number 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 whether they match the pattern. For each keyword in a source file, which matches the pattern or pattern with 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 one, then this 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 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. The identifier values are not added to rows as data, but as metadata. In addition, 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 values are 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 multidimensional data, i.e., to the data of a keyword who store a table of values. Currently, one and two dimensional data are supported. For each dimension, the keyword as a row index identifier that stores the dimension size has to be given. For a two dimensional data, the sizes of one of the dimensions can be given in a table, meaning that for each row/column a custom size can

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 10 of 40
---	----------------------------	--	---


be defined. In addition, the first identifier represents the number of rows, while the second 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 assigned the indexes (0,1,2,3,...), while the rows of two dimensional data get assigned the pairs of indexes ($\langle 0,0 \rangle$, $\langle 0,1 \rangle$,... $\langle 1,0 \rangle$, $\langle 1,1 \rangle$,...).

- 3) Filename 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 that contains 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 (see an example of an XML file in [2]).

A database table may store data from several groups of source files, where each group of source files contain 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, the data from various groups of source files 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 to the existing rows. 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 is 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.

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 11 of 40
---	----------------------------	--	---

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.

2.4 Data insertion in destination Radio archive database

The insertion of data in the Radio archive database is done as follows. At the beginning, the basic data of the input file (such as file name and storage path) is inserted in the main table (i.e., the “file_info” table in Figure 4 and Figure 5). This table also stores the version of the input file. The file version is obtained by checking all the existing files in the database with the same name and determining their maximum file version. Then 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 firstly locking the main table, then reading the maximum file version, 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 the current thread to store all the other data in the database.

When the basic input file data is stored in the main table, the other data is stored 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 (see Section 3), copying the input file in the storage directory, and stopping the transaction.


Note that RDI only stores the data as found in the source files. If some data processing is needed, the database’s stored procedures have to be defined and called after the data is inserted in the database tables (as mentioned in the previous paragraph).

If data were successfully inserted in the database and the input file was successfully copied, the transaction is stopped with the commit and the input file in the input directory is deleted. Otherwise, the transaction is stopped with the 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.

2.5 Linked libraries and dependencies

RDI is implemented using **TANGO Java libraries** (version 1.1.7). Source can be found at <http://sourceforge.net/projects/tango-cs/files/JTango/>

File writing conclusion uses Inotify for Java (version 2.1) and Linux “tar” command. The “tar” command is executed by creating a new process. Inotify for Java is, on the other hand, included as a library. Before using Inotify for Java, it has to be installed on the computer. Next, the path to the installed native libraries has to be set in the appropriate RDI property. RDI then programmatically adds this path during the initialization to the Java library path. The source and the description of the installation process for the Inotify for Java can be found at <https://bitbucket.org/nbargnesi/inotify-java/>

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 12 of 40
---	----------------------------	--	---

It has to be noted that the input files can be created using various commands, such as local *cp* command, remote *cp* on mounted input directory using NFS or Samba file system, or remote commands without mounting the input directory such as *scp*, *ftp*, *smbclient* and *rsync*. To detect file writing conclusion, several approaches (in addition to Inotify for Java) were tested and the results were as follows:

- Linux *fuser* command was used to detect when the file is not opened by any process. This command fails with *cp* on remote NFS or Samba directory, *ftp*, and *smbclient*.
- Inotify for Java was used to detect “File closed” event. This approach fails with *cp* on remote NFS directory.
- Jnotify⁶ and Watch Service⁷ were used instead of Inotify for Java for detecting “File create” and “File rename” events. Both reject, i.e., miss events when a large number of events occurs.

To sum up, only Inotify for Java (in combination with Linux “tar” command and file size checking) enabled to recognize file writing conclusion.

To **untar** the input files, Apache Commons Compress (version 1.2) is used. Source can be found at https://commons.apache.org/proper/commons-compress/download_compress.cgi

To create a **pool of reusable database connections**, Apache DBCP2 (version 2.1), Pool2 (version 2.3) and Logging (version 1.2) components are used. The Logging component is required and only used by the DBCP component. Sources can be found at <https://commons.apache.org/proper/commons-dbc/>, <https://commons.apache.org/proper/commons-pool/>, <http://commons.apache.org/proper/commons-logging/>

RDI is connected to an (existing and properly configured) **MySQL database** using the JDBC driver for MySQL, called MySQL Connector/J (version 5.0.8). Source can be found at <http://dev.mysql.com/downloads/connector/j/>

The **emails** are sent using Javax Mail library (version 1.5.3). Note that this library requires a properly installed Linux SMTP server whose configuration enables to send emails. Source can be found at <https://java.net/projects/javamail/downloads>


XML files are read using Javax XML Stream library (as part of Java 7). The documentation can be found at <http://docs.oracle.com/javase/7/docs/api/javax/xml/stream/package-summary.html>.

(MB)FITS files are opened and read using **ESO JFITS library** (version 0.94). Source can be found at http://www.eso.org/~pgrasbol/fits_java/jfits.html

It has to be noted that the JFITS library was changed/enhanced due to specific RDI needs as follows. The first letter of a keyword can be a number, which was not feasible in the original JFITS implementation. In addition, the pointers are handled transparently in the enhanced version. For example, if a value is a pointer to a string, the original JFITS implementation returned the pointer. On contrary, the enhanced version returns the actual string.

⁶ <http://jnotify.sourceforge.net>

⁷ *java.nio.file.WatchService*

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 13 of 40
---	----------------------------	--	---

2.6 Installation and configuration

2.6.1 Registration in TANGO

To register RDI in TANGO, the Jive⁸ tool can be used as follows. Select “Edit”, then select “Create server”. In the opened form, the following data has to be written (predefined data, i.e., mandatory values, are in bold, user-selected data are denoted as “<xx>”):

1. Server: **RadioDataImporter**/**<instance_name>**
2. Class: **RadioDataImporter**
3. Device: **<facility>/<device>/<device_instance>**

When starting RDI, the **<instance_name>** has to be passed to it as an argument.


Before RDI can be started, a set of (device) properties has to be set. To that end, a configuration file storing property-value pairs can be created and load with the Jive tool by clicking “File” and then “Load property file”. In addition, proper access rights have to be set for the input directory, temporary directory and storage directory/directories. This can be done by, e.g., executing the command “`chmod ugo+rw directory_name`”. Moreover, proper access rights have to be set for the used databases. To that end, appropriate commands have to be executed such as MySQL command “`grant all privileges on schema_name.* to 'user_name'@'localhost';`” and “`grant select on mysql.proc to 'user_name'@'localhost';`” (the last command enables to execute stored procedures).

In addition, RDI can be started only after the datamodel database is properly configured, i.e., the proper values are inserted in the datamodel database tables.

2.6.2 Device properties

- Data for establishing the connection to the datamodel database: *DBHost* (string), *DBPort* (int), *DBUser* (string), *DBPassword* (string), *DBSchema* (string), *DBType* (string, e.g., MySQL, Oracle, etc.)
- Names of tables in the datamodel database (for its description see Section 3):
DBDestinationTable (string), *DBInstrumentTable* (string), *DBInstrumentConfigurationTable* (string), *DBConfigurationTable* (string), *DBDatabaseTable* (string), *DBDataTable* (string), *DBStepDataTable* (string), *DBSourceFileTable* (string), *DBOtherDataTable* (string), *DBPrecedentTable* (string), *DBContentIdentifierTable* (string), *DBRowIndexIdentifierTable* (string), *DBFilenameIdentifierTable* (string), *DBFilenameIdentifierTableTable* (string), *DBProcedureTable* (string)
- *AutoStart* (boolean): if true, the On command is executed after the Init command
- Several properties for sending emails: *CriticalMailFrom* (string), *CriticalMailTo* (string[]), *CriticalMailHost* (string, the IP address of the computer that will send the emails on the critical errors), *CriticalMailSubject* (string), *CriticalMailText* (string)

⁸ <https://www.tango-controls.org/downloads/tools/>

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 14 of 40
---	----------------------------	--	---

- *DefaultInstrument* (string): the name of the default (warning) instrument
- *FileCheckingSleepTime* (int): the sleep time in milliseconds between two file writing conclusion checkings
- *InotifyJavaNativeLibPath* (string): path to the native libraries of the Inotify for Java
- *InstrumentList* (string[]): the list of instruments whose files will be processed. The input files of other instruments will not be processed and will be assigned to the default instrument
- *TmpPath* (string): the directory where the tar files will be (temporary) untarred
- *WatchPath* (string): the input directory where the Inotify for Java listens for the new input files
- *FileProcessingWorkerNumber* (int): the maximum number of threads for file processing that can be active at each moment. The sets of active and non-active (waiting) threads are maintained by Executor Service 2
- *FileWritingWorkerNumber* (int): the maximum number of threads for file writing conclusion checking that can be active at each moment. The sets of active and non-active (waiting) threads are maintained by Executor Service 1

2.6.3 Automatic startup configuration


TANGO can be used to automatically start RDI at system initialization. To that end, the Astor⁹ tool can be used. By default, Astor shows the database to which it is connected. If already properly configured, it also shows the controlled hosts. However, if it does not control the host where RDI is configured, this host has to be added manually. This is done by selecting “Command” and then “Add a New Host”. Afterwards, the host name (the name of the computer) has to be inserted. Finally, the device servers’ path has to be defined. For example, the path to the “bin” folder of the TANGO installation can be inserted, e.g., “/usr/local/tango/bin”. This folder is then used to store the executables of the TANGO servers. Therefore, the RDI jar file has to be copied in this folder. Note that the jar file must have exactly the same name as the TANGO server, e.g., without the “.jar” extension. In addition, proper access rights have to be set for this file, e.g., by executing the command “chmod ugo+rx file_name”.

Although RDI can be started by calling the jar file from shell or at system initialization, it is more convenient to use a script to configure and execute RDI. This script can also be stored in the TANGO bin folder and is then called at startup by TANGO (if properly configured in Astor). In this case, the script has to have exactly the same name as the TANGO server, while the jar file that is called by the script must have a different name than the TANGO server.

2.6.4 Custom logging

RDI writes the descriptions of events occurred during file processing in the log. When a file is processed successfully, only an info is written. Otherwise, a description of the error is stored in the log. By default, the TANGO log is used. If preferred, a custom logging can be used. To that end, a log configuration file with the name “logback.xml” has to be created. This file can be stored in the

⁹ <https://www.tango-controls.org/resources/documentation/tools/>

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 15 of 40
---	----------------------------	--	---

same folder as the RDI jar file. In this case, the option “ -classpath . ” has to be specified when starting the jar file.

Note that if the automatic startup of RDI is configured with Astor, then in addition to RDI jar file, also the logback.xml has to be stored in the .../tango/bin directory. The linux user that starts the Astor (e.g., the “controls” user) has to have the privileges for these two files (obtained, e.g., by executing “chmod ugo+rw file_name”). Note also that the same user has to have privileges to write the log files in the log directory, defined with the logback.xml.

2.6.5 Applying data access policy

The data stored in the database are publicly available without restrictions. On the other hand, the input files (whose paths are stored in the database) are publicly available only after a predefined time with respect to the creation time. In the meantime, the input files are available only to the primary investigator, who is the data owner. The name of the primary investigator is stored in the “piname” column in the “data_file” table (see Section 5).

To apply such a policy, the “public” column is defined in the “data_file” table. Its value is “0” by default. A bash script is used to set public to “1” when a predefined time after the input file insertion elapses. This script is executed by the Cron job scheduler¹⁰. To that end, the script must be registered in the Crontab by executing “crontab -e” and adding a row to the opened document, such as “0 11 * * * /path/to/the/script.sh”. In this example, the script is executed every day at 11:00.

¹⁰ <https://en.wikipedia.org/wiki/Cron>

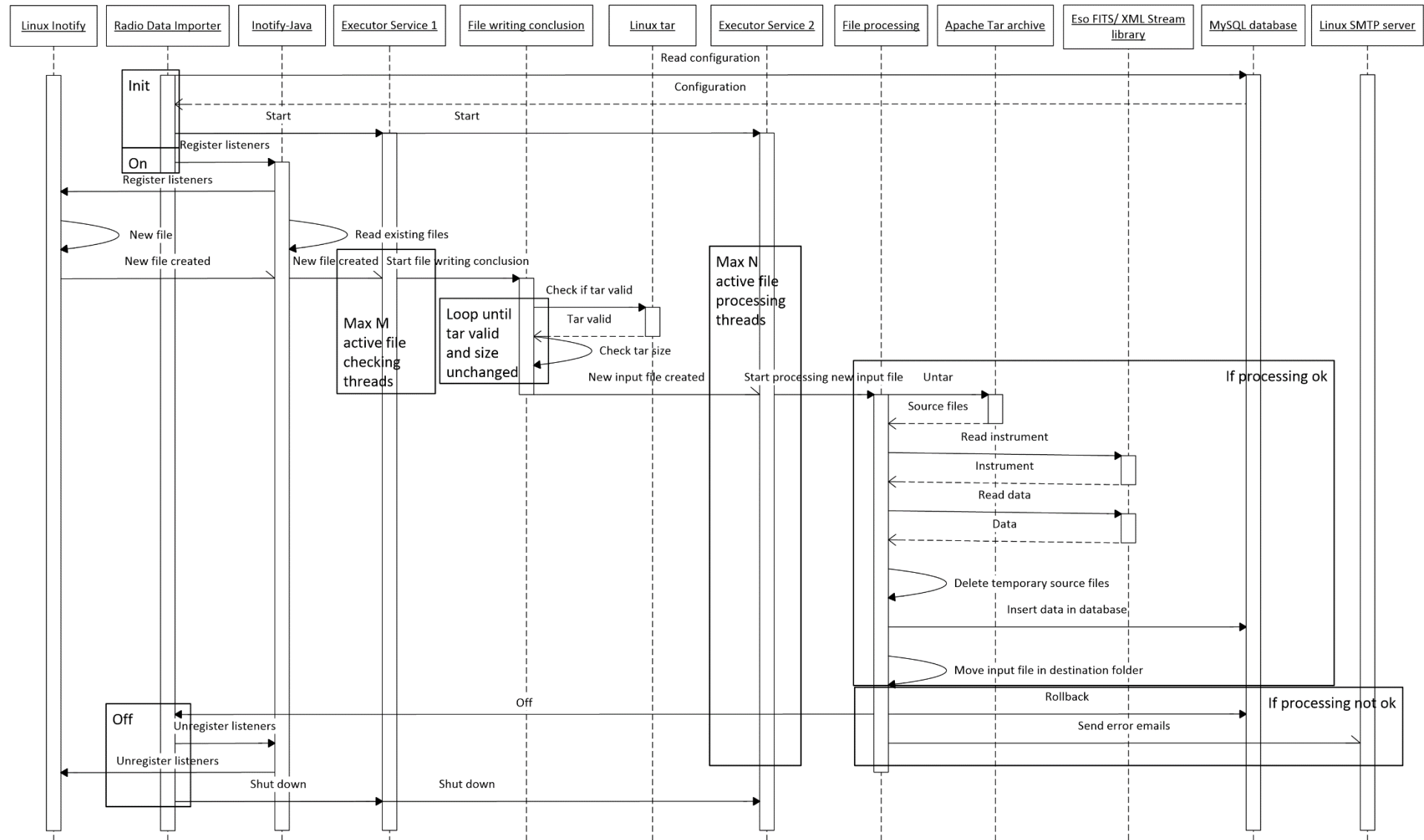



Figure 1: Radio Data Importer sequence diagram



	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 18 of 40
---	----------------------------	--	---

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 (see Sections 4 and 5). To that end, it stores the data of instruments, the data of destination 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 RDI configuration is mainly based on the MBFITS file format. An MBFITS file structure contains several files such as GROUPING, FEBEPAR etc. For more information on the MBFITS file structure, see [1].

To check whether a connection to a datamodel database is still open and active (see Section 2.1.4), the following SQL command is used: "SELECT destination_id FROM destination".

The following sections describe the groups of tables of the datamodel database (note that tables are marked in bold).

3.1 Instrument definition


The datamodel database defines one or more **destinations**, where each of them defines an instance of the Radio archive database (i.e., stores connection data including schema name and type, e.g., MySQL, Oracle, etc.) and a storage directory.

Each destination can store data of various **instruments**. Each instrument is uniquely defined with its name. In addition, it has a directory name, where the input files are stored, and the name of the main table in the Radio archive database (*file_info_table_name*). Moreover, it stores a data type, which is a user-defined name of the type of data that are stored in the input file. For example, if the input file stores an MBFITS, the data type can be "fits-mb". When inserting the input file in the Radio archive database, the data type of the input file instrument is copied in the FILE_INFO table. This enables to distinguish the input files based on their type.

There can be several **configurations** that define the data to be read from the source files and stored in the Radio archive database. Each (but the default, "warning", instrument) has associated an **instrument_configuration** that is used to read data from each input file coming from this instrument. An instrument configuration defines the keyword that stores the instrument name (foreign key *data_id_instrument_name*), and the keyword that stores the observation date (foreign key *data_id_date_obs*).

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.

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 19 of 40
---	----------------------------	--	---

3.2.1 Definition of tables

The tables that store the data from the source files are listed in the **database_table** in such sequence that no precedent table needs a foreign key from any subsequent table. Besides the table name, it stores the `result_id` as the name of the primary key column. When the row is inserted in the Radio archive database, the value of the primary key is automatically created by the database with 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**. 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.

3.2.2 Definition of data of source files


The table **source_file** stores the path and name patterns that are used to search for the source files that are of the same type, i.e., contain the 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, this data is 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 the data with the same FEBE value have to be concatenated in the same row.

3.2.3 Data definition

The table **data** stores the definition for 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
current implementation supports only "header"+integer and "data"+integer for the purpose of processing the FITS and MBFITS file; when processing the XML files, the file part is ignored

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 20 of 40
---	----------------------------	--	---

- keyword pattern pri and sec
firstly, the primary keyword has to be checked, if not exist, then secondary keyword is checked; if multiple instances of one keyword are feasible, e.g., FEBEn in GROUPING, then it has to be defined as a pattern, e.g., FEBE.*
- the name of the column in the Radio archive database where datum will be inserted
- 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 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 not found); if not countable all keywords are tested if they match the pattern
- are_multiple_dimensions: used only when file part contains "data"; 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 this datum. For example, CHANNELS from ARRAYDATA store the number of rows that could be inserted in a table related to channels
- mandatory: if true and this datum not exist in the source file, then the input file is assigned to the default instrument
- read_as_single_datum: if true then all the bits that store the value of this datum are read as 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

The data that are stored in each table are defined in **step_data**.


3.2.4 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") 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**

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 21 of 40
---	----------------------------	--	---

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 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, where the row index is equal to the FEED index. If the datum represents multiple dimensional data, then one *row_index_identifier* is defined on only one dimension of these data. Consequently, multiple *row_index_identifiers* can be defined on the same multiple dimensional data, each index on 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 and 3; etc.

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 as described in Section 2.3. Note that this identifier is not inserted/stored in the datamodel database.

An example of identifiers is as follows. GROUPING stores multiple FEBEn, while multiple FEBEPAR files store data related to specific FEBEs. In this case, a content identifier has to be used for GROUPING and a filename identifier has to be used for FEBEPAR to determine the FEBE of each row in order to concatenate the correct rows for the insertion in a FEBE-related table.

3.2.5 Configuration use case

Examples of usage of a datamodel database (in connection to the Radio archive database described in Sections 4 and 5) are as follows. Note that the tables of the datamodel database are in bold, while the tables of the Radio archive database are capitalized.

When a row is inserted in the DATA_FILE table, an auto-generated ID is returned. This value is associated with the name stored in the result_id in **database_table** for the row describing the data of the DATA_FILE table. If another table such as FEBE requires result_id as the foreign key to the

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 22 of 40
---	----------------------------	--	---

DATA_FILE table, the row describing the data for FEBE has to be connected to the table **other_data**, where **other_data.name** must have the same value as **result_id** of the row in the DATA_FILE table. Therefore, when inserting data in the FEBE table, all the rows of the connected tables (through the table **precedent**) have to be checked whether they contain the ID with the name **other_data.name**. In the case of insertion of a FITS/MBFITS file, only one row is inserted in the DATA_FILE table. This row ID (its name is stored in **result_id**) is used as the foreign key in the FEBE table.

However, when a table is connected (with foreign keys) to another table, other than DATA_FILE, or when an XML file is processed, the procedure is more complex. For example, ARRAYDATA is connected to BASEBAND and SUBSCAN_FEBE (this is stored in the **precedent** table). Each row in SUBSCAN_FEBE contains the ID of the FEBE and the SUBSCAN. In addition, each row in the BASEBAND contains the ID of the FEBE and the BASEBAND. When reading data for ARRAYDATA, the following filename identifiers (stored in table **filename_identifier_table**) are obtained: subscan ID, FEBE ID, and baseband ID. Next, the objects already inserted in BASEBAND and SUBSCAN_FEBE have to be checked to find the ones with the same subscan ID, FEBE ID and baseband ID. When the two objects are found, one for each table, their IDs are read (i.e., BASEBAND_ID and SUBSCAN_FEBE_ID) and inserted in the ARRAYDATA row. The names of the IDs to be inserted in ARRAYDATA row are stored in **other_data.name**.

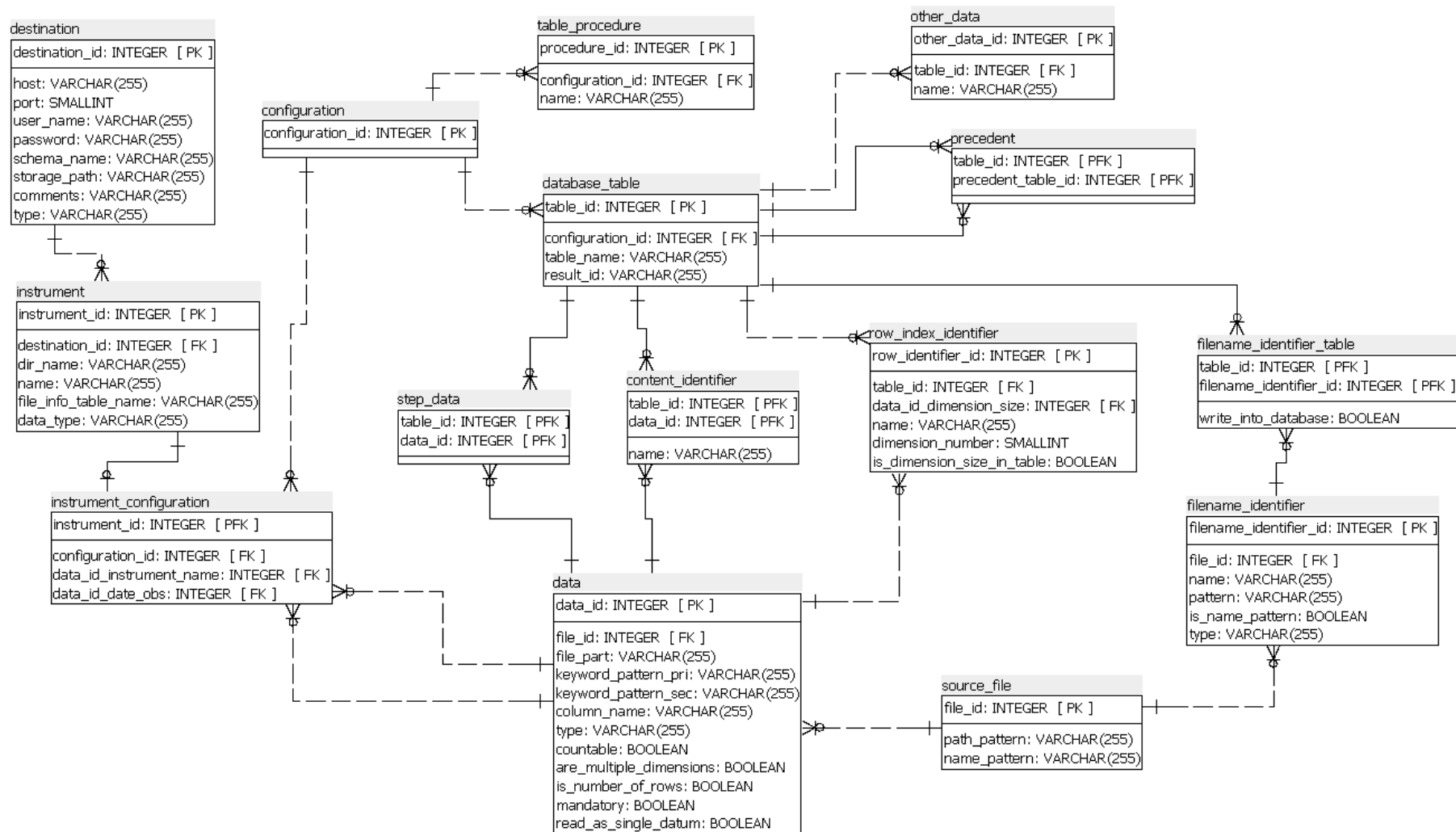



Figure 3: Datamodel database

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 24 of 40
---	----------------------------	--	---

4. General MBFITS database

4.1 Overview

A general MBFITS database is used as a baseline for the creation of the Radio archive database. A Radio archive database (described in Section 5) is then used to store the data retrieved from the source files based on configuration stored in the datamodel database (described in Section 3). The configuration enables the user to select any subset of data from the source files that are interesting and thus have to be stored in the Radio archive database. In order to store the subset of data, the Radio archive database 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 structure of data 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 [1]. 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.

4.2 Description of tables and data cardinality

Table 1 shows the symbols that are used to describe the MBFITS database. The tables used to store the MBFITS data are described in Table 2. The MBFITS database including all the needed tables and a subset of data columns is shown in Figure 4.

Table 1: Symbols for describing the MBFITS database

SYMBOL	DESCRIPTION	VALUE	INDEX
N_{FEBE}	Number of FEBEs	$ \langle i_{FEBE} \rangle - FEBEPAR (=SCAN.NFEBE)$	i_{FEBE}
N_{SCAN}	Number of subscans	SCAN.NSUBS	i_{SCAN}
$N_{BAND}^{i_{FEBE}}$	Number of used basebands for FEBE	$\langle i_{FEBE} \rangle - FEBEPAR. NUSEDDBAND$	i_{BAND}
$N_{FEED}^{i_{FEBE}}$	Number of feeds for FEBE	$\langle i_{FEBE} \rangle - FEBEPAR. FEBEFEEED$	i_{FEED}
$N_{USEFEED}^{i_{FEBE}, i_{BAND}}$	Number of used feeds for baseband for FEBE	$\langle i_{FEBE} \rangle - FEBEPAR. NUSEFEED [i_{BAND}]$ Repeated in $\langle i_{FEBE} \rangle - ARRAYDATA - \langle i_{BAND} \rangle.$	$i_{USEFEED}$

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 25 of 40
---	----------------------------	--	---

		NUSEFEED	
$N_{CHANNEL}^{i_{SCAN}, i_{FEFE}, i_{BAND}}$	Number of channels for a baseband of a FEFE during a subscan	$i_{SCAN}/<i_{FEFE}>-ARRAYDATA-<i_{BAND}>.$ CHANNELS	$i_{CHANNEL}$
$N_{MONITOR}^{i_{SCAN}}$	Number of monitor raw data for a subscan	MONITOR. BINARY_TABLE	$i_{MONITOR}$
N_{PHASE}	Number of phases	SCAN.PHASEn	i_{PHASE}
$N_{PHASE}^{i_{FEFE}}$	Number of phases per FEFE	$<i_{FEFE}>-FEBEPAR. NPHASES$	i_{PHASE}



	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 26 of 40
--	-------------------------------	--	---

Table 2: MBFITS database tables and data cardinality

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}	$\langle i_{FEBE} \rangle$ -FEBEPAR	
SUBSCAN	N_{SCAN}	i_{SCAN} /MONITOR	Each subscan corresponds to one subfolder, where the subfolder name represents the SUBSCAN name
SUBSCAN_FEBE	$N_{SCAN} N_{FEBE}$	$i_{SCAN}/\langle i_{FEBE} \rangle$ - 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	$\sum_{i_{FEBE}=1}^{N_{FEBE}} N_{BAND}^{i_{FEBE}}$	$\langle i_{FEBE} \rangle$ -FEBEPAR binary table only	Only data related to basebands, e.g., BOLCALFC. Names are stored in USEBAND, however, they are not needed since only baseband indexes are used as identifiers
ARRAYDATA	$N_{SCAN} \sum_{i_{FEBE}=1}^{N_{FEBE}} N_{BAND}^{i_{FEBE}}$	$i_{SCAN}/\langle i_{FEBE} \rangle$ - ARRAYDATA- $\langle i_{BAND} \rangle$	One row for each baseband of each FEBE for each subscan
FEED	$\sum_{i_{FEBE}=1}^{N_{FEBE}} N_{FEED}^{i_{FEBE}}$	$\langle i_{FEBE} \rangle$ -FEBEPAR binary table only	Only data related to feeds, e.g., FEEDOFFX

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 27 of 40
--	-------------------------------	--	---

BASEBAND_FEED	$\sum_{i_{FEBE}=1}^{N_{FEBE}} \left(N_{FEED}^{i_{FEBE}} N_{BAND}^{i_{FEBE}} \right)$	< i_{FEBE} >-FEBEPAR binary table only	Only data (two-dimensional tables) related to basebands and all feeds, e.g., APEREFF
BASEBAND_USED_FEED	$\sum_{i_{FEBE}=1}^{N_{FEBE}} \left(\sum_{i_{BAND}=1}^{N_{BAND}^{i_{FEBE}}} N_{USEFEED}^{i_{FEBE}, i_{BAND}} \right)$	< i_{FEBE} >-FEBEPAR binary table only	Only data (two-dimensional tables) related to basebands and used feeds, e.g., BESECTS. Names of used feeds are stored in USEFEED
CHANNEL	$\sum_{i_{SCAN}=1}^{N_{SCAN}} \left(\sum_{i_{FEBE}=1}^{N_{FEBE}} \left(\sum_{i_{BAND}=1}^{N_{BAND}^{i_{FEBE}}} N_{CHANNEL}^{i_{SCAN}, i_{FEBE}, i_{BAND}} \right) \right)$	$i_{SCAN}/<i_{FEBE}>$ - 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	$\sum_{i_{SCAN}=1}^{N_{SCAN}} \left(\sum_{i_{FEBE}=1}^{N_{FEBE}} \left(\sum_{i_{BAND}=1}^{N_{BAND}^{i_{FEBE}}} N_{CHANNEL}^{i_{SCAN}, i_{FEBE}, i_{BAND}} N_{USEFEED}^{i_{FEBE}, i_{BAND}} \right) \right)$	$i_{SCAN}/<i_{FEBE}>$ - ARRAYDATA- < i_{BAND} >, only DATA from binary table	The feed order is the same as in <FEBE>-FEBEPAR.USEFEED
MONITOR_DATA	$\sum_{i_{SCAN}=1}^{N_{SCAN}} \left(N_{MONITOR}^{i_{SCAN}} \right)$	$i_{SCAN}/$ 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

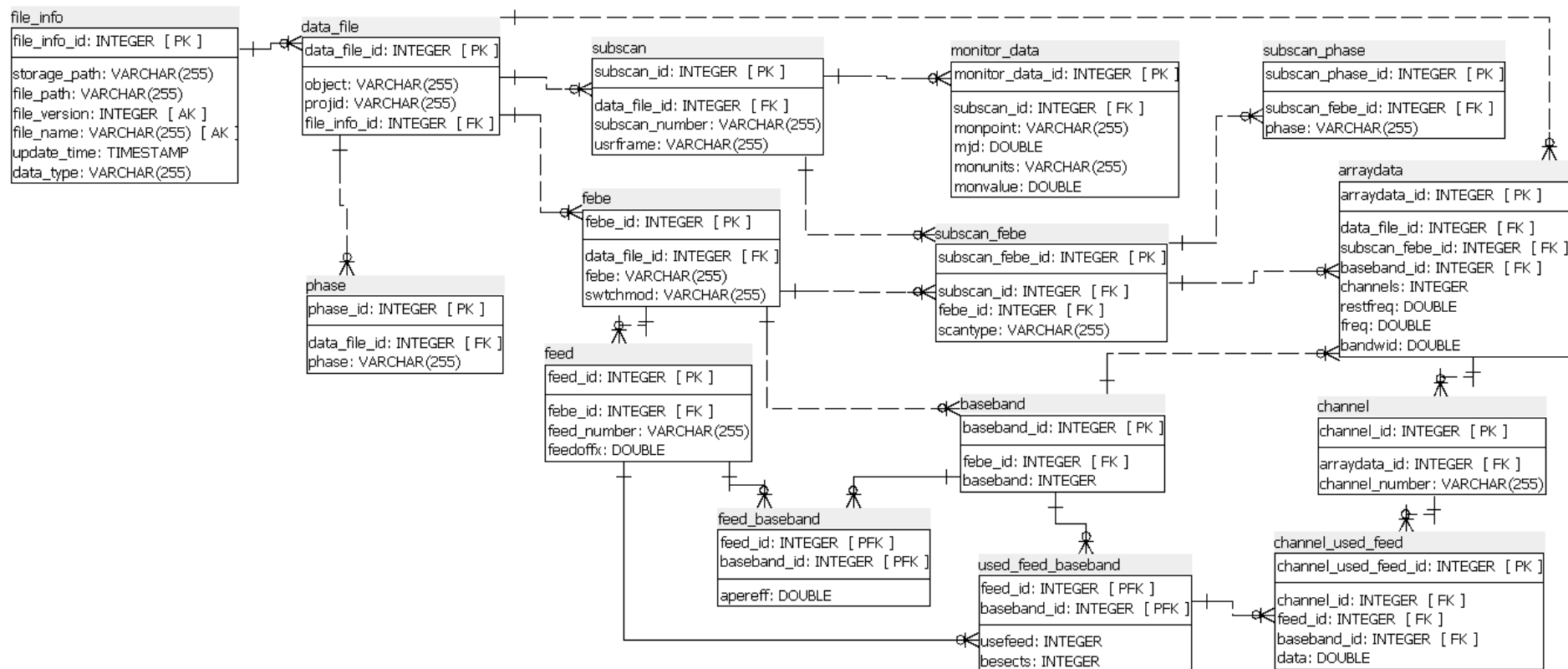



Figure 4: MBFITS database including all the needed tables

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 29 of 40
---	----------------------------	--	---

5. Radio archive database

This section presents the Radio archive database that is a subset of the general MBFITS database described in Section 4. Data that are required to be stored in this database are described in [2].

Note that the presented database is a common 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.

In addition, one format has a common keyword for *febe* while another has two keywords (for *frontend* and *backend*). Therefore, all three data can be stored in database.

The *freq* column denoting the frequency also differs with respect to the type of input data (it is either the minimum or the average frequency). The database enables to store both of them. Afterwards, a database's stored procedure is used to properly calculate the minimum and the maximum frequency, and store them in *min_freq* and *max_freq*.

The column *freq_bin* has not a corresponding value in the MBFITS files. Its value is calculated with a stored procedure after all the data of an input file have been inserted in the database.

Note also that some input files are not valid and therefore the data cannot be read. In this case, a row is inserted only in the main table called *file_info*. This row has no row associated in the *data_file* table. This enables to recognize the input files with issues.

An addition table (called *schedule*) is added to this database 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 observation data, and an input file storing the log and the schedule. 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 moves them in the storage folder and inserts their names in the *schedule* table. Note that these two instances of RDI have to 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, *schedulename* in the *data_file* table has to be matched with the *file_name* in the *schedule* table.

To check whether a connection to the Radio archive database is still open and active (see Section 2.1.4), a new table has to be created. This is required since all the tables can be empty, thus existing tables are not suitable for the checking purpose. The new table is created and populated as follows:

```
CREATE TABLE validation (validation int(11) NOT NULL,PRIMARY KEY (validation));
INSERT INTO validation(validation) VALUES (1);
```

The statement for checking the connections to the Radio archive database is then "SELECT validation FROM validation".

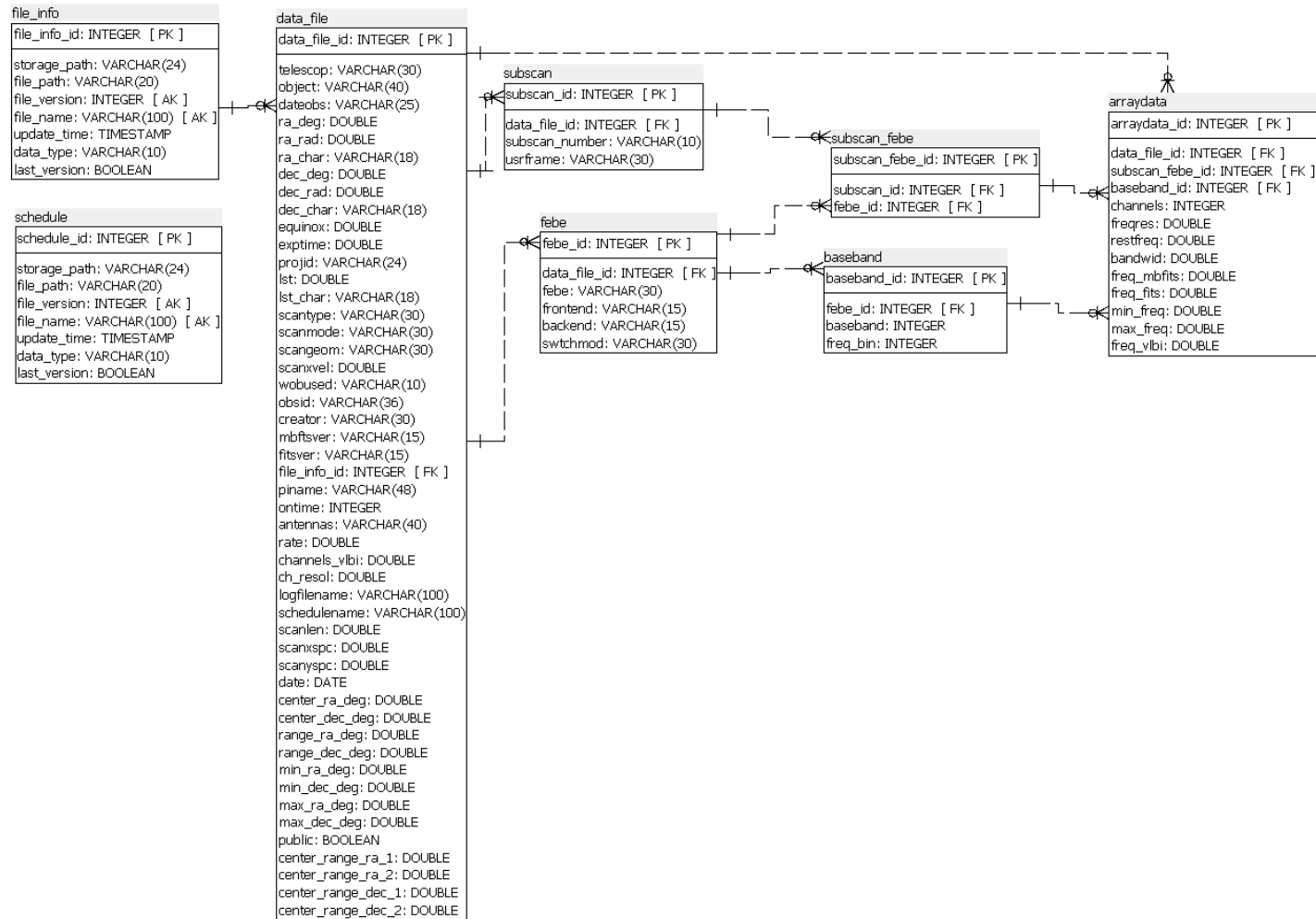



Figure 5: Radio archive database

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 31 of 40
---	----------------------------	--	---

6. Database querying

The data are stored only in the Radio archive database therefore only this database is queried. To speed-up the queries, two approaches are applied. Firstly, all the data that should be calculated during querying, are calculated in advance and stored in the database. Consequently, there is no need to calculate these data at each query and these data are easily indexable. Secondly, indexes are used on various columns and combinations of columns to speed-up the most frequent queries. The following sections describe how queries are built, how data are indexed and how the queries in the form of stored procedures can be called/executed.

6.1 Data preprocessing and query building

The following data are calculated during the insertion of input file data in the database. Minimum and maximum frequencies are calculated with respect to the type of input files (FITS or MBFITS). For MBFITS, *freq_bin* is calculated from *channels*, and *frontend* and *backend* are parsed from *febe*. The date of type “date” is obtained from the observation date of type “varchar” in order to speed-up the queries where initial and/or final dates are given by the user.

The data can be public or private (see Section 2.6.5 on data access policy). When public, all the information is given. On the other hand, when private, the input files, i.e., the paths to the input files, are hidden to all the users except the primary investigator (identified with the *piname* column in the *data_file* table), while the other data stored in the database are always public.

When the schedules/log files are also requested by the user, the *data_file* table (the *schedulename* column) is joined with the *schedule* table (the *file_name* column), where the paths to the schedules are also subject to the data access policy.

When searching for strings, the search is performed as “LIKE %value%”. The only exceptions are the *data_type*, where “LIKE value%” is used, and the *piname*, where equality is used.

When searching for date from and to, the user-given dates are firstly converted into the date type. Similarly, the *lst* parameter is firstly converted into seconds.


The query has to return only the last version of each file. Therefore, it has to return only those files, where *last_version* is true.

The frequencies are stored in ranges in the database, and also searched within a given range. Consequently, the query searches for all the input files where the following condition is true:

```
user_max_f >= db_min_f AND user_min_f <= db_max_f
```

Note that this formula is also used when searching only in one RA-DEC direction, e.g., when the user searches only by DEC. To this end, RA (right ascension) and DEC (declination) are postprocessed as follows. Firstly, both are converted into decimal degrees. Secondly, the range, minimum and maximum RA and DEC of the observation are calculated with respect to the scan mode as follows:

```
IF scan mode IS 'OTF' THEN db_range = scanlen
```

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 32 of 40
---	----------------------------	--	---

```

ELSE IF scan mode IS 'RASTER' THEN db_range = scanlen * scanspc
ELSE db_range = 0

```

```

db_min = db_center - db_range/2, db_max = db_center + db_range/2

```

However, when the user simultaneously searches by both RA and DEC, the query is more complex due to the search for intersections between circles and rectangles. More precisely, the data in the database are obtained by observing/scanning a rectangle in the RA-DEC two-dimensional space. When a user defines the RA-DEC parameters, he/she also defines the range of the circle around the RA-DEC center. Therefore, the data of all the observations whose observing rectangles intersect with the user-given circle have to be returned. The procedure for intersection determination is as follows¹¹ (see also Figure 6).

```

boolean intersects(circle, rect)
  distance.x = abs(circle.x - rect.x)
  distance.y = abs(circle.y - rect.y)
  if (distance.x > (rect.width/2 + circle.r)) return false
  if (distance.y > (rect.height/2 + circle.r)) return false
  if (distance.x <= (rect.width/2)) return true
  if (distance.y <= (rect.height/2)) return true
  distance_sq = (distance.x - rect.width/2)2 + (distance.y - rect.height/2)2
  return (distance_sq <= (circle.r2))

```

- 1) The first pair of lines calculate the absolute values of the x and y difference between the center of the circle and the center of the rectangle. This collapses the four quadrants down into one, so that the calculations do not have to be done four times. The image shows the area in which the center of the circle must now lie. Note that only the single quadrant is shown. The rectangle is the grey area, and the red border outlines the critical area which is exactly one radius away from the edges of the rectangle. The center of the circle has to be within this red border for the intersection to occur.
- 2) The second pair of lines eliminate the easy cases where the circle is far enough away from the rectangle (in either direction) that no intersection is possible. This corresponds to the green area in the image.
- 3) The third pair of lines handle the easy cases where the circle is close enough to the rectangle (in either direction) that an intersection is guaranteed. This corresponds to the orange and grey sections in the image. Note that this step must be done after step 2 for the logic to make sense.
- 4) The remaining lines calculate the difficult case where the circle may intersect the corner of the rectangle. To solve, compute the distance from the center of the circle and the corner, and then verify that the distance is not more than the radius of the circle. This calculation returns false for all circles whose center is within the red shaded area and returns true for all circles whose center is within the white shaded area.

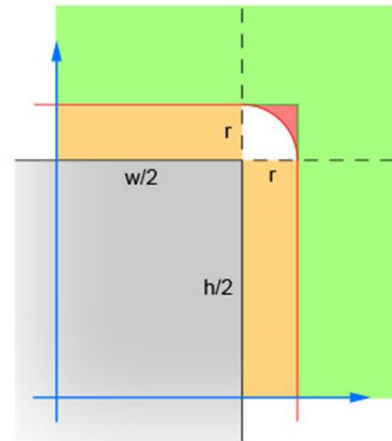



Figure 6: Intersection between rectangle and circle

¹¹ <http://stackoverflow.com/questions/401847/circle-rectangle-collision-detection-intersection>

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 33 of 40
---	----------------------------	--	---

This procedure can be stored as function and called during the query for each row. However, in this way the database management system cannot optimize the query, therefore it is better to avoid calling functions in a query. To that end, this procedure has to be transformed in an inline condition (to be used within the query). Note that the user defines a circle, therefore,

$\text{circle.x} = \text{user_ra}$, $\text{circle.y} = \text{user_dec}$, and $\text{circle.r} = \text{user_range}$.

On the other hand, database stores rectangles, therefore,

$\text{rect.x} = \text{db_center_ra}$, $\text{rect.y} = \text{db_center_dec}$, $\text{rect.width} = \text{db_range_ra}$ and $\text{rect.height} = \text{db_range_dec}$.

The inline condition is then defined as follows:

$\text{abs}(\text{user_ra} - \text{db_center_ra}) \leq (\text{db_range_ra}/2 + \text{user_range})$ and
 $\text{abs}(\text{user_dec} - \text{db_center_dec}) \leq (\text{db_range_dec}/2 + \text{user_range})$ and
 $(\text{abs}(\text{user_ra} - \text{db_center_ra}) \leq (\text{db_range_ra}/2) \text{ or } \text{abs}(\text{user_dec} - \text{db_center_dec}) \leq (\text{db_range_dec}/2) \text{ or } (\text{abs}(\text{user_ra} - \text{db_center_ra}) - \text{db_range_ra}/2)^2 + (\text{abs}(\text{user_dec} - \text{db_center_dec}) - \text{db_range_dec}/2)^2 \leq \text{user_range}^2)$

However, such a condition is not suitable for indexing. This is because, e.g., columns are subtracted, added, divided, made absolute, at the end compared. To enable indexing, it is better to compare a column to a value (without subtraction, addition, division etc.). To that end, parameters *center_range_ra_1*, *center_range_ra_2*, *center_range_dec_1*, and *center_range_dec_2* are calculated in advance (and stored in the database) as follows.

$\text{center_range_1} = -\text{db_center} - \text{db_range}/2$, $\text{center_range_2} = -\text{db_center} + \text{db_range}/2$


Afterwards, the inline condition is redefined as follows:

$\text{center_range_ra_1} \leq (-\text{user_ra} + \text{user_range})$ and $\text{center_range_ra_2} \geq (-\text{user_ra} - \text{user_range})$ and
 $\text{center_range_dec_1} \leq (-\text{user_dec} + \text{user_range})$ and $\text{center_range_dec_2} \geq (-\text{user_dec} - \text{user_range})$ and
 $((\text{center_range_ra_1} \leq -\text{user_ra} \text{ and } \text{center_range_ra_2} \geq -\text{user_ra}) \text{ or } (\text{center_range_dec_1} \leq -\text{user_dec} \text{ and } \text{center_range_dec_2} \geq -\text{user_dec}) \text{ or } (\text{abs}(\text{user_ra} - \text{db_center_ra}) - \text{db_range_ra}/2)^2 + (\text{abs}(\text{user_dec} - \text{db_center_dec}) - \text{db_range_dec}/2)^2 \leq \text{user_range}^2)$

Note that there are still some calculations to be done in the last line of this condition. This is due to the fact that this line cannot be transformed in a more appropriate form. However, this line is checked only in a small number of cases, therefore is expected that it will not slow down the search significantly.

6.2 Data indexing

Indexing the Radio archive database is a hard task due to several facts. The most complex requirement for indexing is that the user can query the database by limiting the values of each subset of around 40 data columns present in the database. However, it is not feasible to simply index each of these columns. Consequently, it is reasonable to index only those columns that are queried in the majority of queries. Note that from this point of view, index management has to be a permanent process, where periodical analyses regarding the most frequent queries have to be done and the most frequently used parameters have to be selected for adjusting the set of indexes.

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 34 of 40
---	----------------------------	--	---

Although the selection of indexes has to be periodical, some things have been discovered already during the development process. For example, it is reasonable to index the *data_type* in the *file_info* table to speed up the search of specific data types (FITS, MBFITS and XML). In addition, the *piname* should be also indexed for the searches when the primary investigator is logged in the system.


RA, DEC and frequency should also be indexed, since these are the most relevant (meta)data of the observations. Note that it is reasonable to index only the columns that appear in the WHERE condition of the queries. Therefore, it is not reasonable to index, e.g., *freq_fits*, but *min_freq* and *max_freq* should be indexed. Min and max data always appear in pair in a query, therefore can also be indexed in pair, i.e., with one index on both data columns. Note that the indexing sequence has to be the same as the sequence of these columns in the WHERE condition. For example, an index can be defined on the following two columns: *center_range_ra_1*, *center_range_ra_2*.

Although MySQL can use more than one index for a table, the preliminary tests show that more than one index were never used. Consequently, special care has to be given to the queries where several conditions are defined, since only one index is used in practice. For example, foreign keys are indexed by default and these indexes are very suitable when joining tables, especially when the database is relatively structured as in the case of the Radio archive database. Therefore, it is reasonable to combine querying columns and foreign keys in common indexes. For example, the following two indexes can be defined for frequencies: (*min_freq*, *max_freq*, *data_file_foreign_key*) and (*data_file_foreign_key*, *min_freq*, *max_freq*). In this case, first index will be used when the data selection begins on the *arraydata* table, while the second index will be used when the data selection begins on some other table that is joined with the *arraydata* table afterwards.

The *file_name* in the *schedule* table should also be indexed since it is used to join the *schedule* table with the *data_file* table. However, since this is an outer join, there is no need to index *schedulename* in *data_file* (i.e., the joining column). This is due to the fact that the appropriate rows are firstly find in *data_file* (and connected tables), and only afterwards for each of these rows the *schedule* table is checked if the schedule data exist.

There are some columns that do not need indexes. For example, the *public* column is never used in WHERE condition. The *storage_path*, *file_path*, *file_version* and *update_time* columns in the *file_info* table are also never queried. One exception is *last_version* in *file_info*, which is in the WHERE condition. However, during the preliminary tests, indexing this column slowed down the queries, when the conditions were defined in such a way that no row was returned. This is due to the fact that if the selection is firstly done on, e.g., *data_file* and no row is returned, no selection is then done on *file_info*. However, by defining an index on *last_version*, a selection was firstly done on *file_info* using this index, then the rows were joined with *data_file*, and only afterwards it was determined that no row should be returned. Such a sequence significantly slowed down the queries. An additional argument for not indexing *last_version* is the fact that a large majority of files will (presumably) have *last_version* equal to true, therefore an index on this column will not significantly reduce the selected rows.

Regarding the *schedule* table, only *file_name* is queried, therefore, no other column should be indexed. The *data_file* table is not queried on the following columns: *dateobs*, *ra_deg*, *ra_rad*, *ra_char*, *dec_deg*, *dec_rad*, *dec_char*, *lst_char*, *logfilename*, *scanlen*, *scanxspc*, *scanyspc*, and

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 35 of 40
---	----------------------------	--	---

public. The *subscan* table is not queried on *subscan_number*. The *febe* table is not queried on *febe*. The *arraydata* is not queried on *freq_mbfits*, *freq_fits* and *freq_vlbi*.


Note that a significant limitation for defining the indexes represents the fact that the strings must be searched as “LIKE %value%”. In practice, this means that the indexes on the string/varchar columns will not be used. The only exception are *piname* and *data_type* that are searched with the equal sign or “LIKE value%”, therefore the indexes can be used/defined.

6.3 Query execution

The interface for executing previously described queries consists of a set of database’s stored procedures as described below. Each procedure returns a table of data.

The main stored procedure is *get_data*. It has to be called with the following parameters:

- *[set of booleans]*: a boolean has to be given for each datum that has been extracted from the input files and stored in the Radio archive database. Each boolean determines whether the corresponding datum should be returned by the procedure, i.e., whether the datum should be included in the “SELECT” part of the SQL statement.
 - *[set of values]*: the query value has to be given for each datum that has been extracted from the input files and stored in the Radio archive database. This value is used to limit the returned data to a subset containing only the specified value of the datum. This is achieved by adding the value in the “WHERE” part of the SQL statement. If NULL, then the results are not limited to a specific value of a datum, i.e., nothing is added to the “WHERE” part of the SQL statement regarding this datum.
 - Four additional parameters:
 - o *add_schedule_table [boolean]*: it determines whether the results should contain also the paths to the schedule file, i.e., an additional column with the schedule path.
 - o *logged [boolean]*: it determines whether the user is logged in the system. If he/she is logged, his/her private data will also be returned (in addition to publicly available data). Note that if *logged* is TRUE, the user name (i.e., the name of the primary investigator, *piname*) should also be given (but it is not mandatory, see the exception below). By taking into account the data access policy (see Section 2.6.5), the following scenarios can be applied:
 - if *logged* is FALSE, then only paths to publicly available input files are returned, while paths to private files are not returned
 - if *logged* is TRUE and *piname* is given, only the data of the selected primary investigator are returned including the paths to all (his/her) input files
 - for administrators only: if *logged* is TRUE and *piname* is not given, then the returned data contain the paths to all (private and publicly available) input files and these data are not limited to a specific primary investigator only.
- In all three scenarios, all the data in the Radio archive database (except the file paths) are always returned.
- o *count [boolean]*: if true, then the resulting table of data is not returned, but its rows are counted, and only the number of rows is returned.

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 36 of 40
---	----------------------------	--	---


- *number_of_rows* [int]: if NULL, all the rows are returned. Otherwise, only the first *number_of_rows* rows are returned, i.e., the “LIMIT” part is added to the SQL statement.

The *get_data* stored procedure returns the following table of data:

- if *count* is TRUE, it returns only one value, i.e., only the number of rows
- if *count* is FALSE, it returns a table of data containing the following columns:
 - file id
 - file path or EMPTY string, depending on the fact whether the user is logged and the input file is public
 - optional column returned only when *add_schedule_table* is TRUE: schedule path or EMPTY string, depending on the fact whether the user is logged and the input file is public
 - a set of columns, each corresponding to a datum from the Radio archive database, which was selected to be returned by the procedure when this procedure was called. The columns are in the same sequence as they appear in the vector of the *get_data* input parameters. Obviously, the columns of the data that were not selected to be returned are not present in the returned table.

There are four additional stored procedures that only upgrade the interface. More precisely, these procedures only call the *get_data* procedure with preset parameter values and return its results. The differences between the *get_data* procedure and the four additional procedures are as follows:

- The *get_radio_data* procedure: the same data (i.e., columns) that are limited are also returned. Consequently, no [set of booleans] has to be given (see the set of *get_data* input parameters described above). In addition, *count* is set to FALSE and is not present as an input parameter.
- The *get_radio_data_selected_columns* procedure: *count* is set to FALSE and is not present as an input parameter.
- The *count_radio_data* procedure: it is used to count the number of rows. The rows contain the same data (i.e., columns) that are limited. Consequently, no [set of booleans] has to be given (see the set of *get_data* input parameters described above). Since only the number of rows are returned, the following parameters are preset: *add_schedule_table* = FALSE, *count* = TRUE, *number_of_rows* = NULL. Consequently, these three parameters are not present as input parameters.
- The *count_radio_data_selected_columns* procedure: since only the number of rows are returned, the following parameters are preset: *add_schedule_table* = FALSE, *count* = TRUE, *number_of_rows* = NULL. Consequently, these three parameters are not present as input parameters.

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 37 of 40
---	----------------------------	--	---

7. Database replication

Database replication was developed based on the existing programs, namely data exporter, data importer, metadata exporter and metadata importer. Metadata exporter and importer are used to copy the database data from the master database (where metadata exporter is running) to the replication database (where metadata importer is running). In addition, data exporter and importer are used to copy the files (whose links are stored in the master database) from the master server (where data exporter is running) to the replication server (where data importer is running).

The description of the existing programs can be found at:

http://ia2-doc.oats.inaf.it/projects/metadata_exporter/wiki

http://ia2-doc.oats.inaf.it/projects/metadata_importer/wiki


http://ia2-doc.oats.inaf.it/projects/data_exporter/wiki

http://ia2-doc.oats.inaf.it/projects/data_importer/wiki

Here are described only the upgrades needed due to specific radio data replication requirements. Note that data exporter and importer remained basically the same since only some bugs were fixed. Note also that data importer requires two database tables for logging of issues during the input file transfer (as described at http://ia2-doc.oats.inaf.it/projects/data_importer/wiki).

Metadata exporter and importer were, on the other hand, enhanced as follows. Regarding the metadata exporter, support for data from multiple database tables was added. More precisely, the original implementation enabled to transfer data from only one table with arbitrary number of columns. To that end, metadata exporter stored the data from the database in an internal multiple-row structure before sending it to metadata importer. Note that neither the row structure nor the row size was defined in advance. In addition, the structure and the size can vary among the rows of one instance of the multiple-row structure. This enabled to reuse the existing structure for the storage of data from multiple database tables, where each table contains a subset of data of the input file(s). More precisely, when radio archive database is taken into account, data of (one) input file are stored in the *file_info*, *data_file*, *febe*, etc. database tables (see Figure 5). Therefore, when data of an input file is to be replicated, data from all these tables have to be simultaneously copied to the replication database using one message containing one multiple-row structure. To that end, the existing structure was upgraded by adding the name of the target database table at the beginning of each row.

In order to correctly read the data from the master database and populate the multiple-row structure, a set of SQLs has to be defined in advance and stored in the master database. The SQLs have to be executed in the predefined order, where each SQL reads the data from one database table. The order is crucial for the values of the foreign keys of a row since all the referenced values (primary keys in the connected tables) have to be already present in the database when the row is to be inserted. Note that the SQL for reading the data from the main table (containing generic data on the input files such as file name, file path etc.) is already hard-coded, while all the other SQLs have to be stored in the master database table. This table consists

	Radio Data Importer Report	Document No.	IA2-ARC-035 / OATS Technical Report n. 206
		Issue/Rev. No.	1.0
		Date	16 February 2016
		Page	38 of 40

of the *id* column for determining the sequence, and the *sql_statement* column, storing the SQLs. These SQLs contain three parameters:

1. @SCHEMA is the schema in the master database, which contains the data
2. @TS1 is the update time of data, stored in the main table, from which the data are retrieved; it has to be used in SQLs as not inclusive, i.e., *main_table.update_time* > '@TS1'
3. @TS2 is the update time of data, stored in the main table, to which the data are retrieved; it has to be used in SQLs as inclusive, i.e., *main_table.update_time* <= '@TS2'

Regarding metadata importer, the insertion procedure was changed in order to retrieve the name of the database table from the first value of each row of the metadata exporter message.

The previously described enhancements enable to transfer the new data from the master database to its replication, i.e., they manage the *INSERT* statements. However, some data are changed in the master database after some time (by applying *UPDATE* statements). More precisely:


1. The *last_version* in the main table changes to false, when a new input file with the same *file_name* is inserted in the database.
2. The *public* in the *data_file* table changes after a predefined period of time according to the data access policy (see Section 2.6.5).

These updates are not transferred with messages from the master database to the replication, but they are achieved with the implementation/deployment of the (same) update procedures as implemented/deployed on the master server. More precisely:

1. The *last_version* is updated with the metadata importer by checking when a new row is inserted in the main table, retrieving the *file_name* and updating *last_version* of all other rows with the same *file_name*.
2. The *public* is updated by deploying the same script on the replication server as deployed on the master server. For the script deployment procedure, see Section 2.6.5.

Metadata exporter and importer, and data exporter and importer are implemented as TANGO servers. Their adaptation for the Radio archive database replication resulted in the following changes of their TANGO properties:

1. Metadata exporter: added *AuxSchema* and *AuxTable* (both string), which are used to determine the schema/table that stores the SQLs used for obtaining the data from the master database. Note that SQLs for reading the data from all but the main table have to be ordered, i.e., an SQL that reads foreign keys should be executed after all the SQLs that read the referenced keys. Note also that no *AuxTable* is needed if data is transferred only from the main table. The last case corresponds to the data from the *schedule* table (see Figure 5), where no data expect generic input file information is stored.
2. Metadata exporter: added *ExportedSchema* (string) that stores the name of the schema where the data are stored.
3. Metadata exporter: changed *ExportedTables* (array of string), now storing only the name of the database tables (without the schema), where the order of tables has to be the same as the order of SQLs in *AuxTable*. Note that the array of tables has to have an additional table,

	Radio Data Importer Report	Document No. Issue/Rev. No. Date Page	IA2-ARC-035 / OATS Technical Report n. 206 1.0 16 February 2016 39 of 40
---	----------------------------	--	---

i.e., the main table, in the first position. Consequently, the second table in the array corresponds to the first SQL in *AuxTable*, the third table to the second SQL, the fourth table to the third SQL, etc., and the last table to the last SQL.

4. Metadata importer: *DatabaseTable* changed to *DatabaseTables* (array of strings) that has to contain the same array of table names as the *ExportedTables* property of the Metadata exporter.

The importers and exporters are implemented in C++, thus they have to be compiled before the deployment. The compilation requires an existent installation of the Nadir libraries (for details, see <http://ia2-doc.oats.inaf.it/projects/tango/wiki>). Note that paths to some Nadir libraries (specified in the Main file) may have to be updated since, e.g., 32- instead of 64-bit version of the SOCI library¹² is installed in the deployment environment.

An example of the deployment of data exporter and importer, metadata exporter and importer, and RDI for handling the FITS, MBFITS and XML input files, and schedules is shown in Figure 7.

¹² <http://soci.sourceforge.net/>

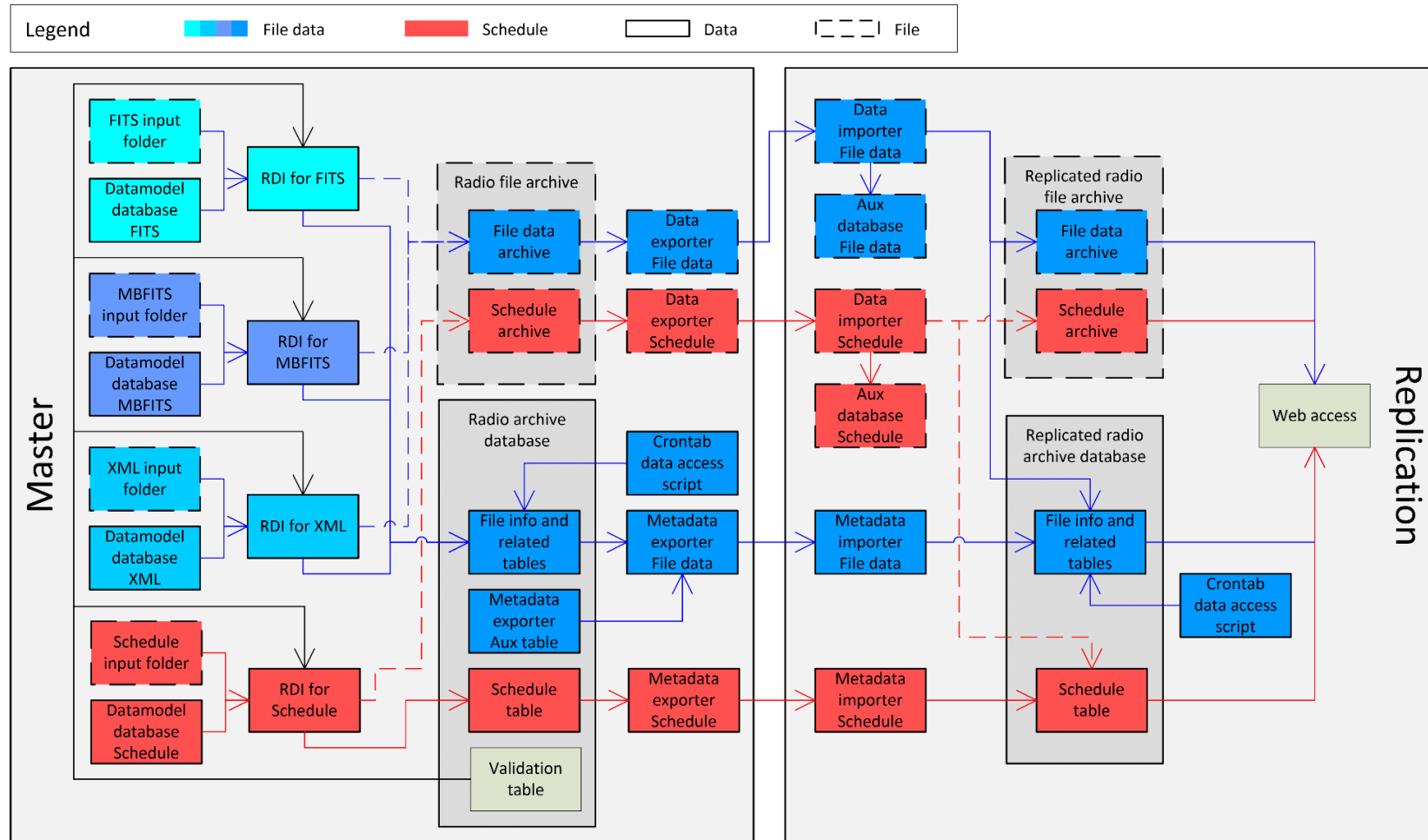


Figure 7: Example of the deployment of the RDI and replication TANGO servers