**Version Notes:**

**4.1.6:** Implements:

- Logging: Full logging support for all modules defining 3 types of logger: FrontEnd Logger, BackEnd Logger and DB Logger. Each Logger can define separate Error and Debug files.

- Full support for background threads with classes IntervalTimer and IntervalFunctions. Background threads are limited in number and in the Intervals they can use in order to minimize potential re-entry and simultaneous resource access.

- ProgActivity class: Programmed Activities are defined as object instances of the ProgActivity Class. This class encapsulates all that’s required for a programmed activity, including the handling and administration of the ActivityTrigger class.

Interfaces are defined between Activity and ProgActivity classes to determine whether an executed activity is a closing activity for a ProgActivity object, among other interfaces

- Implemented setRecords() using executemany(). Execution time for 1 record drops from 12.5 millisec to 5 millisec for 1 record. Since executemany() performs all the writes in 1 single operation, with 1 commit at the end, writes are much faster**: 5000 records in 3 msec!!.**

- Implemented class UploadBuffer() to manage database synchronization. See synchronization section .

- **Activities. Reentry / Simultaneous access. – 12May23**

Activity objects are Singletons (only 1 instance created in each class). In order to support addressing and managing system objects, a property is defined for each Activity Object where the Animal, Tag, Person, etc is passed as an attribute (outerObject attribute).

This is a *dynamic attribute* for the execution of the activity (*it changes so must be set at runtime with every call to the Activity methods*).

Situation:

* Activity objects are Singletons.
* Activity Classes have numerous methods that can be called in parallel (simultaneous access to the class) by the many threads running or recursively (nested access, call within a call, in an Activity method)
* Hence, the Activity Class must enable a mechanism to preserve the class’ dynamic attributes for each of its methods when they are called simultaneously from the same thread or from different threads.

*Any dynamic attribute for Activity Singletons must be place in a LIFO structure (list preferably). The code within the activity methods must retrieve the last value of the attribute and store it in such a way that reentries to the same code preserve the original attribute for the method they were initialized for.*

*outerObject is the sole candidate (at the moment) for this treatment. -> \_\_outerAttr = []*

*At the time, looking into using a decorator within the class to manage the LIFO \_\_outerAttr list.*

Trying this at the moment:

@staticmethod  
def \_dynamic\_attr\_wrapper(func):  
 *""" Wrapper to remove from \_\_outerAttr list the outerObject appended by the @property methods that perform the  
 Activity function calls  
 To decorate any function that is called via @property (inventory, category, assignTag, status) """* @functools.wraps(func)  
 def wrapper(self, \*args, \*\*kwargs):  
 a = func(self, \*args, \*\*kwargs)  
 if self.\_outerAttr:  
 with self.\_\_lock:  
 self.\_outerAttr.pop() # Removes the last object from \_\_outerAttr list in Activity object.  
 return a  
 return wrapper

Important: This ‘stack’ approach is not safe as there are NO guarantees that the thread-switching processes managed by the OS will allow for func to complete its excution and peform an orderly pop() on the \_outerAttr stack before another thread-switch happens.

This is resolved with a dict. See notes in the *Multithread access to SQLite* section, at the end.

Information on JOURNAL\_MODE:

<https://www.sqlite.org/pragma.html#pragma_journal_mode>

**- Initial, subtle changes to the UploadBuffer execute() method:** attempt to implement an abstract calling interface by inserting the execute() function for the object in the object structure, thus passing the function to be called along with the object.

**4.1.7:**

TODO:

- **Implement Geo objects and TM classes** with their respective registers and functions.

- **Implement Sequences.**

- **Implement Activity Factories** for Activity class and Animal class, in order to extend the activities and types of animals to classes defined through the user interface. Use type(cls, parent, {}) to implement the factories. *Already implemented in v 4.1.6.*

For this:

**a.) In Animal Activity Class (and PersonActivity, etc):**

1) Define method \_\_init\_subclass\_\_(cls) in AnimalActivity as follows:

def \_\_init\_subclass (cls, \*\*kwargs):

super().\_\_init\_subclass (cls, \*\*kwargs)

cls.register\_class() # A list of all Activity Classes that implement activity objects (inventory, # status, etc).

**b.) In Animal (and Person, Tag, etc.):**

See <https://realpython.com/python-callable-instances/#creating-callable-instances-with-__call__-in-python>

Using the class register from above (it’s set () returned by get\_class\_register()), run a loop to:

1. Create Activity objects (using a factory method).
2. Instantiate caller methods for each object as instances of the MethodFactory class.
3. Implement the @property decorator for each caller object using exec.

class ActivityMethod(MethodFactory):  
 *""" Specific implementation of a methods generator for Activity Classes that use the outerObject feature """* def \_\_init\_\_(self, obj): # obj is an instance of type InventoryAnimalActivity, StatusAnimalActivity, etc.  
 self.\_\_activityObj = obj  
 super().\_\_init\_\_()  
  
 def \_\_call\_\_(self, item\_object): # Implementacion especifica de los detalles de la llamada.  
 self.\_\_activityObj.outerObject = item\_object # itemObj es Animal, Tag, etc.  
 return self.\_\_activityObj

1. Creating the Activity objects. -> Trivial. Use cls (params…) to create inventoryObj, statusObj, etc. Just not with these names but in a list: 1 object per AnimalActivity defined in class\_register:

activityObjects[0] = InventoryActivityAnimal ()

activityObjects[1] = StatusActivityAnimal ()

activityObjects[2] = CatetoryActivityAnimal (), etc.

for j in AnimalActivity.get\_class\_register():  
 try:  
 activityObjList.append(j()) # Creates InventoryAnimalActivity, StatusAnimalActivity singleton objects.  
 except (AttributeError, TypeError, Exception) as e:  
 pass

1. Implement the activity caller function using a MethodFactory class:

for j in activityObjList:  
 try:  
 if j.\_decoratorName: # se salta los None  
 setattr(j, j.\_decoratorName, ActivityMethod(j)) # creates callable inventory in InventoryAnimalActivity,  
 method = getattr(j, j.\_decoratorName) # status in StatusAnimalActivity, etc.  
 # Aplica decorator @property. Necesario porque convierte method a class property. Si no, no puede  
 # method = property(method) # acceder a los metodos de InventoryAnimalActivity, etc.  
 setattr(Animal, j.\_decoratorName, property(method)) # Crea atributo inventory, status, etc. en class Animal.  
 except (AttributeError, TypeError, Exception):  
 pass

1. Define the activity caller name:

settattr(self, self.decoratorName)

self is class Animal, Person, etc. an instance of an item object.

decoratorName comes from DB, tied to the AnimalActivity in question.

caller\_name will be *inventory, status, category, etc*

1. Create an ActivityMethod object that when assigned invokes its \_\_call\_\_() method and initializes outerObject.

caller\_name = MethodFactory(activity\_object)

*Already implemented in v 4.1.6.*

Interesting how all this works together:

1. Activity Singleton objects are created in step 1 using classes listed in the activity\_class\_register. (initialized beforehand by the method \_\_init\_subclass\_\_() )
2. A for loop runs through all those objects and creates a method object for each of them. These method objects are of class ActivityMethod and they implement the \_\_call\_\_() function that makes the object callable. This method assigns the underlying caller object (Animal, etc) to \_\_outerAttr.

def \_\_call\_\_(self, item\_object): # Implementacion especifica de la llamada.  
 self.\_\_activityObj.outerObject = item\_object # itemObj: Animal, Tag,   
 return self.\_\_activityObj

1. Another loop runs through the ActivityMethod objects crated above and:
   1. Creates an attribute in the Activity Classes created in one, assigning it the name defined in their \_decorator attribute.
   2. Assigns the ActivityMethod object to that new attribute.

if j.\_decoratorName: # se salta los None  
 setattr(j, j.\_decoratorName, ActivityMethod(j)) # creates

* 1. Gets the new attribute using getattr() and applies the decorator to it, morphing the ActivityMethod object into a property object, which is absolutely required to make the calls from the Animal, Tag, Person objects.

1. In order this this to work, ***the ActivityMethod callable objects must be turned into properties.*** I learned this the hard way. This is done here:
   1. A new attribute is created in the Animal class using setattr() as below, assigning to it the name defined by \_decoratorName and the value of the callable object turned now into a property object. See below.

dd callable inventory in InventoryAnimalActivity,  
 method = getattr(j, j.\_decoratorName) # status in StatusAnimalActivity, etc.  
 # Aplica decorator @property. Necesario porque convierte method a class property. Si no, no puede  
 # method = property(method) # acceder a los metodos de InventoryAnimalActivity, etc.  
 setattr(Animal, j.\_decoratorName, property(method)) # Crea atributo inventory, status, etc. en class Animal.  
 except (AttributeError, TypeError, Exception):  
 pass

The very interesting fact (to study more) is how the item Obj argument in the original \_\_call\_\_() method is assigned the self argument from the Animal object in a call like this one:

bicho.inventory.set() :

* bicho is of type Animal.Bovine.
* set() is a method that belongs to ActivityInventoryAnimal class.
* inventory is of type property. Its initial type is ActivityMethod. inventory calls \_\_call\_\_()

def \_\_call\_\_(self, item\_object): # Implementacion especifica de la llamada.  
 self.\_\_activityObj.outerObject = item\_object # itemObj es Animal, Tag, etc.  
 return self.\_\_activityObj

In this call, Python correctly assigns

* self to \_\_call\_\_(self) to access the \_\_call\_\_ method in the correct ActivityMethod object (there is one of them for each Activity defined).
* ***self argument from the underlying object (bicho) is correctly assigned to item\_object in \_\_call\_\_()****.*

*This is because how the property() function operates. This very detail allows for this whole construction to work.*

- Later, see if some Class Factories make sense.

- Implement background code for **AsyncBuffer** Objects:

The Implementation of closure for progActivities to be done in the background (including automatic closure), using AsyncBuffers -> *Implemented 13Jun23 using dedicated class ProgActivityCursor.*

The implementation was done using a decorator, the code shown below.

@staticmethod  
def \_\_bufferWrapper(func):  
 @functools.wraps(func)  
 def wrapper(self, \*args, \*\*kwargs):  
 if self.\_\_progActivityBuffer: # Si hay una instancia de AsyncBuffer, manda data al queue.  
 kwargs.update({'outer\_obj': self.outerObject})  
 cur = self.\_\_progActivityBuffer.enqueue((args, kwargs), the\_object=self, the\_callable=func)

# *TODO(cmt): Si espera por cur.result, se pierde toda la ganancia (100 usec va a 15 msec si se espera!)* return cur.result  
 return None  
 else:  
 return func(self, \*args, \*\*kwargs)  
 return wrapper

The idea behind the code is that it can be used for any function in the Activity Class, and any function that is decorated, when called, queues up itself and its parameters on buffer (in the form of a cursor object) and returns immediately (returning None).

Then the buffer handling logic will take the cursors from the buffer using a dedicated, asynchronous thread, and will make the function calls with the parameters passed.

*As of now, \_paMatchAndClose() function is decorated to operate with this asynchronous execution approach.*

2 main issues must be solved with this approach:

* The outerObject call cannot be used in an asynchronous fashion as its updated independently in function calling thread -> A copy of outerObject is passed in function args and the function must make a copy of outerObject internally in its code:

if not outer\_obj: # Si se llama desde wrapper, se debe pasar outer\_obj.  
 outer\_obj = self.outerObject # outer\_obj es variable local.

* With asynchronous execution, the main thread must wait until all the objects in the buffer queue are processed, or data will be lost/corrupted (in the case of \_paMatchAndClose(), the last item in the list of objects found for closure is not written to database).

To solve the problem a flag is implemented that is to be populated by all the AsyncBuffer objects that *do write* to DB using the async\_cursor.

In an orderly shutdown, all the buffers sending data to DB must flush their queue first by doing a stop() and only after they issued the stop() can the db writer (SQLiteQueueDataBase object) stop itself.

*Doing this in a reverse manner causes the writer to stop prematurely and leaving the buffer threads with no ability to complete their db write, which results in a system hangup. Verified in a painful 11-hour debugging session.*

With the writers\_busy flag, the main db writer can verify if any of the db-writing threads is active, and if there is one, the db writer can issue a message and exit without shutting down and wait until no writers are active to process a writer stop() command.

The stop() command effectively:

1. Puts a SHUTDOWN object in the queue so that the queue clears itself and the loop() function exits (quits processing queue objects).
2. Issues a join() to the main thread. This halts the main thread until all the items in the queue have been processed and the loop() function exits. Then, control is returned to the main thread to resume execution

**Full implementation of Notifications** for progActivities. -> Another candidate for an AsyncBuffer(see above AsyncBuffer for closing ProgActivities)

- **Merge SQLiteQuery and SQLiteQueueDatabase in one class** and select read/write code by parsing the SQL string: **Won’t do it for now.** execute\_sql () function is already implemented but Initial tests fail with “database is locked error” in write operations. Need to keep looking.

- **Implement a general purpose AsyncBuffer for memory operations using the existing UploadBuffer**: The idea is that all operations that can be performed asynchronously and detached from the main user thread can run in a separate buffer thread: An instance of the buffer is created for those operations and the main user thread puts() objects in the buffer queue for asynchronous execution

Try to detach the following code, using a dedicated buffer to manage the closure of ProgActivities:

if self.\_supportsPA:  
 if self.\_supportsPA:  
 executeFields = self.\_baseExecuteFields(exec\_date=tblData.getVal(0, 'fldDate'))  
 if isinstance(execute\_fields, dict):  
 executeFields.update(execute\_fields) # execute\_fields adicionales, si se pasan.  
 self.\_paMatchAndClose(idActividadRA, execute\_fields=executeFields, excluded\_fields=excluded\_fields,  
 pa\_force\_close=pa\_force\_close)

By doing this, all the DB access and checks for a ProgActivity to be closed is performed in a separate, asynchronous thread, freeing the front-end thread.

The idea is that:

* Each buffer instance runs on its own thread, allowing for full independence for managing its operation (START, STOP, PAUSE, UNPAUSE).
* Defines the execute() method within the objects so that the writer executes the code and doesn’t have to implement the code there.
* Target objects (called the\_object) and target functions (called the\_callable) are passed as items of the cursor so that when the cursor is unpacked, all the data and the function to processed the data are available for execution.

**10Jun23**.

Look into creating a DataTable from the data in self.\_data, in the execute() method, and inserting into the that table n

# records (n = tbl\_block\_size attribute). Then pack the data with tblPack() and write that data to

# Upoad\_To\_Server. A variable number of DataTable instances will be generated in this way, each of them to be

# written to DB in a for loop. In this way multiple records can be sent over the connection link, along

# with all the table data, to optimze the data transfer.

**29Jun23**.

Triggers. New approach:

Consider Triggers as an attribute of ProgActivities, with 1:1 relationship. In such scheme, a ProgActivity can be defined with or without a trigger. See ProgActivities Chapter for details.

Without a Trigger:

**4.1.8:**

TODO:

* Correction to getRecords() function to accept **None and False** in lists and by itself, as in

*…., fldName = (None, 0, False).* This should translate to SQL string as:

“fldName IS NULL OR fldName in (0)”

When by itself: …., *fldName is None* should translate to SQL string as:

“fldName IS NULL” (This is already implemented)

* Solved by using dbRead(tblName, sqlstr), where sqlstr includes all the conditions required for the SELECT statement.

Synchronization:

22Jul23: Synchronization will be done by Marmot or similar product.

* Implement Repeat Records/Repeat Objects logic, as per the guidelines listed below.

Repeat Record/Object: an object that is created independently in 2 or more databases (linked to different devices usually) that refer to the same physical object (same Animal, Person, Tag, etc).

This can occur if 2 devices that are not replicating their databases instantaneously process the same object (ex.: same Animal) before having the chance to replicate their databases.

This results in 2 distinct database records, each with its own UUID, referring to the same physical object.

Definitions:

* Original Object: Object with the ‘oldest’ (earliest) TimeStamp field.
* Repeat Object: Object with TimeStamp value later in time than the Original object.

**03Sep23:**

* Completed replication management for:
  + lastCategory, lastStatus, lastInventory, lastLocalization
  + Animal, Tag, Person, Device objects
  + myTags in Animal class.
  + myProgActivities: each node processes only its own ProgActivities (ignores replicated ones) and process ALL executed activities (own and replicated) looking for candidates as closing activities for its own ProgActivities.
  + Sdfs
* Implemented logic to create “multiple singleton” SqliteQueueDatabase objects: the logic for “multiple singletons” is that a SqliteQueueDatabase object is tied to a database name and 1 database name only. So 2 objects can be created for 2 different databases, 3 objects for 3 databases and so on. **It’s a pool of write objects, effectively.**

But when an attempt is made to create an object with an existing database, the already existing object is returned from the pool.

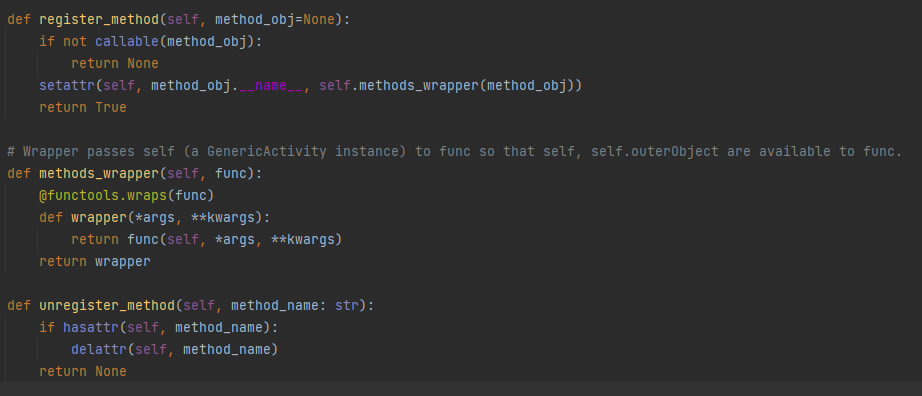
* DB Triggers: Implemented multiple Triggers to enable handling of database replication.

**04Oct23: Generic Activities and Measurements Class implementation.**

**- Generic Activities:** Implemented with 3 base methods: set(), get(), comp() and a basic setting, retrieval and comparison of one 1 user-specified parameter. *Each GenericActivity object is assigned 1 specific Data table from the database.*

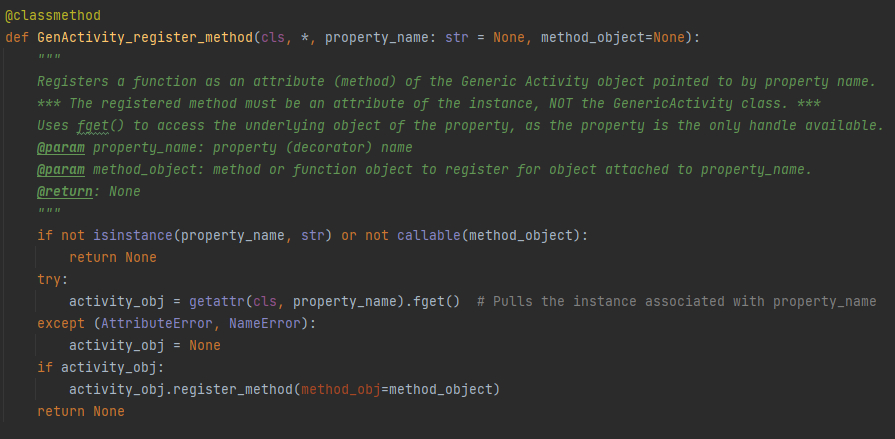
**TODO:**Implement function-registration (register/unregister methods) that register user-defined methods for each individual object of the class. This way, each activity and enhance functionality and operability*. The challenge is to assign registered functions not to the class, but to specific instances of the GenericActivity class. -> Bind register function to the calling object and setattr(target\_method) for the calling object, thus binding it to the object.*

Solved as follows:

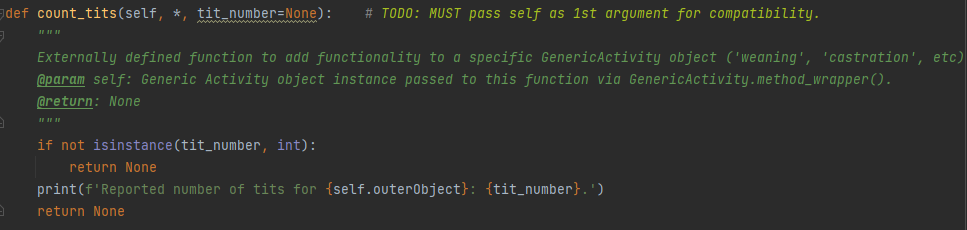


Then, to perform a method registration, must use the property (weaning, castration, branding, etc.) to access the underlying Activity objects for these properties. This is because the object themselves are created via a factory and are not available per se.

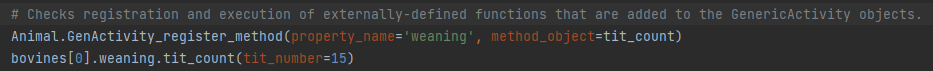
Function fget() in code below does the trick.



Function to register (example): it is important to note that the 1st parameter when defining a function *MUST BE self* (regardless of whether it is used or not).



Registration and execution code (API interface):



*property\_name:* Must be an already defined property name, attached to a GenericActivity object (defined during GenericActivity class initialization).

*method\_object:* The external function code to register with the desired GenericActivity property.

***To define a new Generic Activity****:*

1. *a valid new data table, with a proper structure, must be defined in the database.*
2. *The activity name, its assigned data table and other parameters must be inserted as a new row in the [Actividades Nombres] table.*

- **Measurements**: Measurements class implemented with 3 methods: set(), get(), comp() to allow for easy definition of single parameters to be assigned to Animal class. Example: weight, height, temperature, etc.

***To define a new measurement****: a valid set of values must be inserted in the [Mediciones Nombres] table in the database.*

**4.1.9:**

**4.1.9: 01Dec23**

This version REMOVES ALL the \_\_registerDict memory structures (for Animal, Tag, Person, etc).

\_\_registerDict are of the form {objUID: object, }

This is to avoid going overboard with memory requirements (especially in mobile devices) as the number of objects grow and/or the size of the objects grow.

Instead, a dedicated function ***loadFromDB()*** will return a DataTable with all the active records in tables Animales, Caravanas, etc., to be used ad-hoc by the functions requiring them.

As a reference, a Tag object is 112 bytes in size. Animal is …

A DataTable with 400 Tags is about 3.3K in size. The idea is to use these DataTable objects with active records to create Animal, Tag, Person objects one at a time, process them and discard them instead of creating and keeping all the active objects in memory at all times.

The logic also contemplates that accessing the database just once (to fetch the DataTable with records) and running the class constructors for each object will be efficient in terms of processing time.

In principle, with this change (deprecation of \_\_registerDict memory structures) all the memdata logic for Inventory, Category, localization, status should be no longer required.

The logic that processes replication (function processReplicated) that takes care of

* Duplicate objects resolution.
* ~~Record UPDATES~~
* ~~Record DELETES~~

ONLY the 1st action (duplicates resolution) should be kept. The other 2 are deprecated because memdata fields and \_\_registerDict structures are no longer used.

**IMPORTANT LOOSE NOTES – December 2023:**

* **myTags and myProgActivities** memory structures will be kept as they are for now. They are internal to each Animal/Person object so they share the object’s volatility. While the object is defined, their use simplifies operation.

***22-Dec-2023: MEMORY DATA***

* **MEMORY DATA: A new dictionary is implemented at Animal subclass level (Bovine, Caprine, etc).** The dict has the form

**\_memory\_data: {uid: MemoryData object, }**

* The dictionary is initialized via \_\_init\_subclass\_\_() and is designed to minimize the access to db for oft-queried data, maintaining at the time small memory structures that don’t grow exponentially with the increase in the number of objects managed.

With this, the use of the ‘memdata’ field in database is deprecated.

**15-May-24:**

**A possible use of this could be: the code searches first for an attribute in the \_memory\_data dictionary and if it’s not there (the key doesn’t exist), pulls the data from database.**

The resulting logic then becomes simple: The check above is done by the code (whenever the code deems that appropriate) and the sequence of searching memory first, database second is followed. No flags, no signals or bits keep track of.

The logic for initializing memory data from database must be kept equally simple:

The \_memory\_data keys are looked up in a dedicated \_\_memory\_data\_keys dictionary stored in Bovine, Caprine, etc. The key in that dictionary is the table name and field name where the data should be pulled from for that particular key.

All memory data keys to be pulled from database at \_\_init\_\_ time for simplicity and consistency.

**Initialization of memory data.**

*Important Criterion: memory data* ***cannot be None.*** *A value of None is treated the same as a non-existing key -> The actual value is pulled from db and the value updated in memory. This is because memory\_data is intended to store int, float, str, iterables. None is equivalent to NULL in SQLite, meaning that the data was never initialized, hence the search in database.*

The data is pulled from db tables and 1 field (value field) is allowed to be used as the field to store in memory. The other field used is fldDate (the date the value record was created).

These 2 fields are used in all cases by the loader function.

field is always pulled, using max() function applied to the fldDate field.

If any other data criteria (other than pulling the last record) is required for a specific memory\_data element, additional ad-hoc loader functions will need to be created.

Then load\_memory\_data(cls, tbl: str, fld\_name: str):

Defined in bovine, caprine, device, etc. will attempt to load to memory and initialize dictionaries for all tbl and fld\_name passed.

It will load the correct ones and will fail silently the non-conforming tbl/ fld\_name values.

Then, the querying functions will attempt to pull from memory first, whatever is not found, will then be fetched from database.

**Version 4.2.0. 17-May-24. Memory Data as Activity attribute, and as a Class. A new paradigm.**

*Memory data will be embedded in the Activity code, and fully implemented as an Activity Attribute.*

For that, Activities:

* Will define a flag **supportsMemData**, set to 1 for Activities that do memory\_data.Flag defined within the Activity declaration, no need to use database values.
* will implement methods \_\_memory\_data\_init() / \_\_load\_memory\_data(), \_\_memory\_data\_set(), \_\_memory\_data\_get(), and others as required.
* Will implement uid dictionaries with all active uids in the system and each uid key will hold the memory data structure defined by the Activity. Due to the logic managing \_active\_uids\_dict, this dictionary defined in each Activity MUST keep track of the \_active\_uids\_dict changes.

The \_active\_uids\_dict is handled by the object creation and object duplication logic, which is a complex process, so this dict cannot be copied or assigned by reference to be modified. The logic handling the uids for each Activity will have to access the \_active\_uids\_dict when required, on a read-only basis.

* **The logic to manage the active\_uids for each Activity class can be as follow**s:

1. The objects or classes that invoke Activities will access their \_\_active\_uids\_dict directly. This is done via outer\_obj as follows:

If uid in outer\_obj.get\_active\_uids():

val = self.\_\_active\_uids\_dict[uid].

**The constraint is that Activity classes MUST NOT modify the \_\_active\_uids\_dict in the object’s class.** It’s read-only for them

. A good enough approach is to present only the dict keys to the outside world via the get\_active\_uids() function defined in the Object classes (Bovine, etc):

@classmethod  
def get\_active\_uids(cls):  
return set(cls.\_\_active\_uids\_dict.keys()) # set {animal\_uid}

Then, the actual dict is not accessible.

1. Each Activity Class, during initialization, creates a local \_\_active\_uids\_dict. This internal dict will have the form

\_\_active\_uids\_dict = {uid: memory\_data\_structure, } # class attribute

This will have to be done via \_\_init\_subclass\_\_() function as must create a dictionary locally for each Activity subclass, and make all those accessible from AbstractClassActivity.

memory\_data\_structure: as defined by the Activity class.

Keys in this dict are obtained from cls.get\_active\_uids() call.

ALL uids present in the original dictionary must have an entry in the Activity local dictionaries, with their data set to some value pulled from database or None if no data is found for that particular uid.

1. To access the memory data (get or set):
2. Get the uids set by calling outer\_obj.get\_active\_uids().
3. If uid is found in 1, the code accesses the local \_active\_uids\_dict and attemps to retrieve memory data from there using the uid key.
   1. uid found: retrieves data.
   2. uid not found: there was a new entry in the original dictionary that is not present in the local dict. -> Pulls the required data from the database and adds the uid key with the newly pulled data to the local \_\_active\_uids\_dict.
   3. Returns the data
4. If uid is not found in the original set, the function removes the uid key from the local dictionary, as the uid is no longer present in the original dict.
5. None returned as memory\_data by the get() method means that the database must be queried again to obtain the data, and update the local dict with the data obtained.
6. dfsdfsd

* sdfsdf

sdf

* **INVENTORY:** Starting 4.1.9 it is a memory value, stored in dedicated \_memory\_data dictionary.

**In line with the new use of MemoryData, the Timeout flag is to be moved out of the objects (kept there for ease of reference only, for now) and kept in the \_active\_uids\_dict in memory, along with the Inventory data.**

* **CATEGORIES:** Categories are by nature dependent on animal age (defined by DOB). They will change over time. However, because of the unmanageable number of events, criteria and variables involved in the setting of a category value, it is not practicable to generate categories only by computation.

**It is primarily important because animal date of birth in general is not precisely known and animals are identified on entry by their category.**

Therefore, the logic will be:

* **Categories SHALL NOT BE memory data. The category value is to be computed in real time, everytime.**
  + DOB takes precedence over category setting: When DOB is provided, category is derived from this DOB.
  + Then categories will set by the user will be used, when the user sets one, *at any time* in the life of the animal object. This generally will occur (at minimum) at animal entry (alta) if the dob is not specifically provided.

Whenever a category is set, a dob is computed and set for the animal.

“User-set” categories in this context are categories entered in the system via the category.set() method, which writes the category value in database.

* + An attribute\_set () method is to be implemented to fix/amend category, DOB and other values on animals in case corrections are needed. To be performed with a specific function in order to identify and process corrections through specific method and actions*. Any changes made by this method must also update \_memory\_data info.*
  + When the user sets a category (using category.set() method):
    1. If DOB exists, the function aborts with a message informing that DOB exists and that in order to modify category or DOB, attribute\_set() method must be used.

If DOB doesn’t exist, one is computed from the category. Both DOB and category are recorded in DB.

The set category will also be updated in memory (\_memory\_data dictionary) so that point ii below is consistently executed for all animals.

* + 1. Once the category is set, the time computation logic takes over and will modify the category according to the evolution of time.
    2. Categories can be set by user forward or backward in time with respect to current category using the attribute\_set() method.
  + DOB values must be identified as user-entered or system-generated (when they are computed from a given category value). One way to do this could be the use of a standard value for the hours: minutes: seconds.microseconds of the DOB value itself. Example:

“2023-06-29 22: 22: 22.222222”

The 22: 22: 22.222222 mask is a flag that DOB is system-generated

The mask, for now, is intended for future developments only, as it will not be used in the computation of categories, as described below.

24-Dec-2023:

* + The final version of the logic is:
    - Compute() will use not only the animal age, but the current category also. This is a major change in the logic as the compute() function has to be fully re-written (in a simplified form probably).
    - The logic then is:
      * Categories will be first read from database using the get() method and accessing the latest value for each animal. This is exactly the existing logic.
      * The computation will be made based on the latest animal age and the current category: A category change will be made not on age (and other applicable factors) as absolute values, but in relation to the current category. That is, categories will change ONLY UPWARD of the current category.
      * That’s it. This logic should cover all cases in setting categories within the same animal male/female condition.

This also implies that upon creation of an animal object, with DOB and category are both provided, are set as valid attributes for the object, prioritizing the user-input data as valid data for the animal.

* + The sequence then should be:
    - Categories are read during initialization from database and computations done: for any category change, the new category is set to database.
    - set() Method: sets the category passed to DB as is (after verification for validity). No computations done.
    - get() Method:
      * Reads the category from DB.
      * ***Re-computes the category for current date and age.***
        + *If category changes, calls set() and stores new category in DB.*
      * Returns the updated category.

With this, we can do without the category values in \_memory\_data. No more categories on that dict.

* **26 December 2023:**

**4 Activities must modify the Category of Animals:**

* + - **Parturition (Paricion):** Sets Category to vaca (cow).
    - **Tact (Tacto):** When positive Sets Category to vaca (cow) if it wasn’t set yet.
    - **Weight:** After a certain limit, sets Category to Novillo, Toro, Vaca, etc, based on weight limits for certain categories. => Create Weight Limit column in table Categorias Nombres as JSON: [Weight, Units]
    - **Castration:** Sets Category to Novillito or Novillo.
* **24-May-24 Update. CATEGORIES:** Starting in 4.2.0, Categories ARE memory data. Still, ***they are a computed value: it is computed real time in every call to category.get() using the present value of datetime.***

Being a MemoryData item, and also being a computed value that is computed every time the data is needed, the CategoryActivity functions will have to take on the task of updating the database value of Category everytime it changes (and also, updating in in memory, of course).

**Category data is accessed so often and so widely (for all active Animal objects, basically) by background threads, that it is a must that it be MemoryData.**

**25-May-24 : Due to the fact above, it is also beneficial that the Category set/get functions DO NOT relay on the creation of an Animal object to process the data, that is, the Category changes are performed by the Category object and NOT invoked by an Animal object (using the ‘category’ property).**

**Hence, Category methods get() \_getCategory(), compute() will be re-written as “outerObject independent” so that Animal objects are not created when the Category methods are called.**

**\_setCategory() and compute(), on the other hand, MUST access the outerObject in order to properly set object attributes.**

**The key thing is that get() doesn’t use outerObject and Animal objects are created ONLY when there’s a category change.**

* **02-Jan-2024. (Mexico). The *[Data Animales Actividad Multiple]* table.**

This table is meant to generalize the management of Data type of tables, for multiple Activities, so that when new Activities are defined, there is no need to modify the database structure with the creation of a table specific for the new Activity.

The table keeps 3 data fields for general use:

* + A numeric field (defined as REAL, can be int or float in python).
  + A Text field
  + A JSON field.

Tables with simple data needs can use the numeric and text fields. When more data is required to be stored to database, the JSON field can be used.

In this case, each Activity will define a JSON dictionary in its definition with all the fields required to be stored and managed by the Activity.

Simple, uh?

* **09-Jan-24 (Mexico). Override of functions in GenericActivity class.**

Already defined functions for class GenericActivity (listed in the \_\_reserved\_names structure) can be overridden using a decorator implementation: the newly defined function that uses an already assigned function name must implement a decorator and return an enclosure for execution. Usually, the enclosure will execute at some point the original function (set() function, for example).

All new functions that are not overrides are added as regular function calls using register\_method().

**Enough for vacation work for now…**

* **09-May-24. Implements improvements for JSON processing (orjson, ujson).**

[**https://github.com/ijl/orjson**](https://github.com/ijl/orjson)

Includes specific code for loads() and dumps() functions to facilitate the use of orjson (preferred option) or falling back to ujson or standard json, to improve efficiency of JSON conversions.

* **09-May-24. Implements use of GanadoSQLite\_nr.db** -> non-replicable database (nr) for storing device specific status data.
* **09-May-24. Implements dynamically loadable modules to allow for expansion/updates of code.**

The use of nr database is particularly used for management of dynamically loaded modules. Starting by Tags modules, to enable expansion of tags devices for various technologies. See Dynamic modules section, **Chapter 10.**

* **10-May-24. Implementation of DataBase class.**

DataBase class is used to generalize the use of DataTable class with any and all the databases used by the system. The class contains the database name (db\_name attribute) and implements the internal tables handling methods (load\_tables, loadFields, reloadTables, reloadFields).

Overrides \_\_new\_\_() so that only 1 instance is created for each database used (identified by the database name).

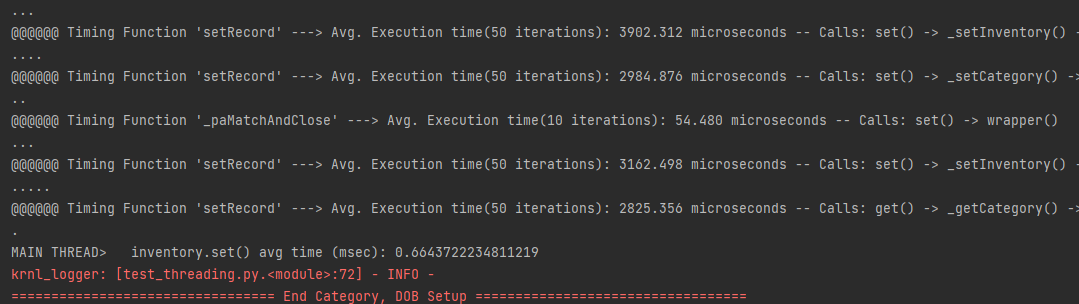
Defined in krnl\_custom\_types.py module.

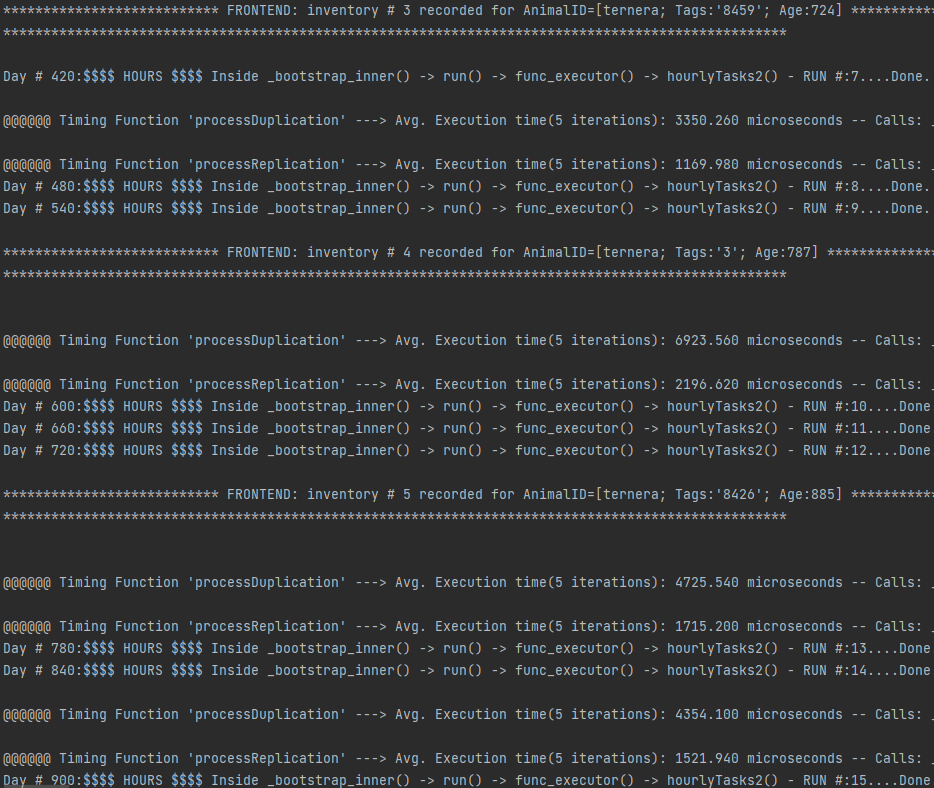
The DataBase class is used by the getFld(), getTbl() methods and for the SQLiteQuery class.

For this, the SQLiteQuery code is updated to handle multiple objects (multiple connections for each open database) in each thread.

* Sdfsd

Timing for setRecord(), \_paMatchAndClose(), processDuplication(), processReplication() with Inventory, Status, Localization as Memory Data. Category NOT Memory Data:





Timing with Category as Memory Data

**4.20:**

**TODO:**

* **Implement CSV import module.**
* **Implement initial version of Sanitation (Sanidad).**
* **Fully implement replication**

Full-fledged test of Marmot replication, ensuring that all replication-handling code, for the items described above on 4.1.8 work as expected.

* **Initialization.**

**Many tasks will require longer initialization times as the databases grow bigger, in particular those reading from Link tables.**

To solve this, use threads:

* + Launch specific functions or methods in dedicated threads and signal busy/completion with Event objects.
  + The objects requesting data from those tasks will check on the completion status and choose to wait or abort when the data they require is not ready.

Memory data has taken on some critical mass. Discussed in detail in Chapter 12.

* **22Jun24: Replication problem: “Unreplicated” objects (Tags, Animals, etc).**

When a tag is read for an object that’s not defined in the database (because of eventually-consistent replication), there will be no information whatsoever on the object (and even on the tag).

The problem may be solved by requesting the user to input some minimum amount of data (based on context) required to operate. Later on, when sync is performed and full data is available for the object, some or all of the minimum data may be discarded or superseded, and some parts may be used.

Start by using a dedicated Status (*No Replicado*) for those objects. That status should determine which activities are allowed and which are not for the object in such status.

* **Sdfsdf**

Test of database replication and handling of duplication

**4.2.1:**

**TODO:**

Full Implementation of Bovine Functions.

Implements DataFrames as a replacement for DataTable across the system.

Implement Activities Memory Data as DataFrames.

**4.2.2:**

**TODO: Implement Notifications**

**4.2.3:**

**TODO: Implement Caprine.**

**4.2.4:**

**TODO: Implement Ovine.**

**4.2.5:**

**TODO: Implement Swine.**

**4.3**

**Chapter 1: Multithread access to SQLite**

22Oct22

There is a re-entry problem with getMaxId(): the function fails to produce the updated index in DB when called from different threads and reentryDict values are repeated, leading to UNIQUE CONSTRAINT FAILED Error when calling asyncCursor \_execute.

For now, the solution SEEMS to be inserting the line

idx = cursor.lastrowid in function setRecord. It might be a problem with commits in the DB. The revised code is:

cursor = writeObj.execute\_sql(strSQL, sqlParams) # TODO: Esta linea va dentro o fuera del lock? -> Estudiar...

idx = cursor.lastrowid

# if strSQL.\_\_contains\_\_('INSERT'): # reentryDict se actualiza SOLO en operaciones INSERT.

if flagINSERT:

counter = tblReentryDict.get(tblName, None) # tblReentryDict = {tblName: [counter, max\_index], }

if counter is not None:

with lock:

if counter[0] == 0:

tblReentryDict.pop(tblName, None) # Si reentry counter volvio a cero remueve tblName del dict.

else:

tblReentryDict[tblName][0] = counter[0]-1 # Si counter > 0, actualiza valor en diccionario

OTRO TEMA: DataTable.setRecords(): Fix the code using getMaxId. Need to use the reentry code using reentryDict.

23Oct22

DataTable.setRecords() re-entrance fixed with the following code:

1) After getMaxId():

rowCount = len(sqlParams)

maxID = getMaxId(self.tblName) # NO PASAR reentryDict aqui, porque hay que manejar multiples records

with self.\_lock:

tblReentryDict[self.tblName] = [rowCount, maxID+rowCount]

2) After writeCursor calls:

cursor = self.\_writeObj.execute\_sql(strSQL, sqlParams)

\_ = cursor.rowcount

if strSQL.\_\_contains\_\_('INSERT'): # tblReentryDict se actualiza SOLO en operaciones INSERT.

counter = tblReentryDict.get(self.tblName, None) # tblReentryDict = {tblName: [counter, max\_index], }

with self.\_lock:

if counter is not None:

if counter[0] == 0:

tblReentryDict.pop(self.tblName) # Si reentry counter volvio a cero remueve key del dict.

else:

tblReentryDict[self.tblName][0] = counter[0] - rowCount # actualiza valor en dict.

We'll see how it goes...

#) DataTable.setRecords(): Removed code using getRecords() to obtain the row index values. Uses existing assignment code to populate list idxRetList which

is used to populate the DB queue of records to send to Server (Upload)

\*\*\* TODO: Improve code in DataTable.setRecords() by using MAXID value in conjunction with rowCount in order to address the index increases and remove the lock

on the cursor.execute command.

\*\*\* TODO: TRRIGGERS: Implement trigger Signatures as a combination of the Trigger conditions to be able to compare triggers in function paCreateTrigger() and

avoid the creation of repeat Triggers.

24Oct22

Re-did write routines, moving get\_max\_id() function inside the cursor classes, removing them completely from the setRecord() and setRecords() function.

Code is simplified, no more re-entry checks and flags. Seems to be working.

With the new write routine, executemany() DOES NOT WORK with AsyncCursor. It will be replaced by loop executions of execute() with the records passed on

to the old executemany().

it DOES WORK however with SQLiteQuery objects but these objects cannot be used in for multithreaded writes.

Using a cursor

cur = self.\_conn.cursor()

cur = cur.executemany(sql, params[0])

will not throw errors, but the writes on DB are NOT performed and the code DOES NOT return from executemany() leaving

the execution flow in an unknown state.

There are some very important advantages to implementing block executions via loop with execute() namely records written to DB (write operations) are

obtained via cursor.lastrowid instead of computing them indirectly. This alone probably justifies dropping executemany altogheter.

25Oct22

AsyncCursor.execute() ALWAYS returns an AsyncCursor object. Then:

To maintain system integrity, AsyncCursor execute() will only call sqlite3.execute() function. In particular, executemany() will not be implemented.

To implement setRecords(), instead of using executemany(), a for loop is generated with a call to execute() in each iteration, with the next element of the input params.

In this way, execute() is made lighter, setRecord() gets lighter and setRecords() is much more consistent, in particular in regards to the indexes

returned, as each index is checked for consistency first (cursor.rowcount > 0) and returned directly from the cursor (cursor.lastrowid).

executemany() can't provide this ability.

A quite interesting finding: setRecords() now implemented by calling cursor.execute() in a loop. It was measured to be 30% - 40% FASTER than setRecord().

That is: a block of 100 records is written by setRecords() in 200 to 250 millisec, or 2 to 2.5 msec per record.

1 record is written by setRecord() in 3 to 4 milliseconds.

While much of the DB writes will have to be done record by record, an effort MUST be made to accommodate the logic to use DataTables and setRecords() as

much as possible, due to this efficiency gain.

26Oct22 - Version 1 (v1) of Multithreading DB writes. Notes and thoughts:

v1 implements the write of DB records by determining the record ID via a call to get\_max\_id() for every record to be written to DB

(INSERT operations only, since ROWID is only not known when INSERTing)

Very important: This approach of calling \_\_get\_max\_id() everytime allows for the implementation of clustered record IDs in

databases, via the assignment of \_\_idxLowerLimit and \_\_idxUpperLimit batches for each DB. get\_max\_id() ensures that all ROWIDs

assigned belong to the assigned batch for each DB individually.

In this way, multiple DBs can run the system on multiple devices and can be sync'd very simply by passing their records "as they are"

without any further processing, as the system avoids all record ID duplication by assigning batches centrally.

The code is in execute() is:

with self.\_conn:

try:

if params:

if tbl is not None and fldID\_idx is not None: # entra aqui solo en INSERT

params = list(params)

params[fldID\_idx] = self.\_get\_max\_id(tbl) + 1 # Actualiza fldID en params.

cur = self.\_conn.execute(sql, tuple(params))

else:

cur = self.\_conn.execute(sql, params)

def\_get\_max\_id() code:

def \_get\_max\_id(self, tbl, \*, row\_count=1):

dbTblName = getTblName(tbl, 0)

dbFldName = getFldName(tbl, 'fldID')

strMaxId = f"SELECT IFNULL(MAX([{dbFldName}]),{self.\_\_idxLowerLimit}) FROM [{dbTblName}] WHERE " \

f"[{dbFldName}] >= {self.\_\_idxLowerLimit} AND [{dbFldName}] < {self.\_\_idxUpperLimit}; "

with self.\_conn:

cur = self.\_conn.execute(strMaxId)

idx = cur.fetchone()[0]

if idx + row\_count > self.\_\_idxUpperLimit: # TODO: ver si corresponde ">" o ">="

self.\_\_idxLowerLimit = self.\_\_idxLowerLimit\_Next # Pasa al siguente batch asignado. Se leen de DB durante init.

self.\_\_idxUpperLimit = self.\_\_idxUpperLimit\_Next # Tabla [Index Batches] asignados a cada dbName

idx = self.\_\_idxLowerLimit

return idx

This is an important design restriction.

Also, due to this design restriction, the maximum value for ROWID assignment (9223372036854775807) will NEVER be reached (See below. This can be enforced

by coding in \_get\_max\_id()). Therefore the Database will behave effectively as if it was running with AUTOINCREMENT = ON.

OCT 26, Afternoon:

An even better, more efficient approach: REMOVE the fldID column from all compose Primary Keys (PK made of multiple columns), and use the ROWID hidden column instead.

With this, the need to updated the fldID column (always redundant, as it will hold the same data as ROWID) is eliminated.

Use ROWID column, along with PK Columns as required, and the fldID Column will not be missed.

This brings the additional benefit that \_get\_max\_id() doesn't need a field name. It will always use ROWID.

And yet another BIG reason to not use WITHOUT ROWID: From sqlit3 documentation: Inserts into WITHOUT ROWID tables are not recorded...

meaning that lastrowid is not set when inserts are done on WITHOUT ROWID tables.

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* What to do when using AutoIncrement is permitted \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* #

If the restriction of fixed lower/upper record IDs DIDN'T exist, an alternative approach could be used to remove the use of

\_get\_max\_id() for INSERT operations:

ON INSERT, in setRecord() / setRecords() functions:

1) Set fldID=None (SQLite converts later the None to NULL)

2.a) If the table is autoincrement (autoIncrementFlag = 1), set/leave parameter db\_fldID = None

2.b) IF the table is NOT autoincrement (autoIncrementFlag = 0), set parameter db\_fldID = DB Name for fldID field (func getFldName())

2) Perform the INSERT operation passing parameters to execute() function

3) in execute() function:

3.1) Perform INSERT.

3.1.1) if db\_fldID == None -> return cursor if cursor.rowcount > 0 else Error

3.2) if db\_fldID != None, save cursor from 3.1.1 in local variable.

3.2.1) Generate sqlString with db\_fldID = cursor.lastrowid from last INSERT

3.2.2) execute UPDATE to populate field db\_fldID = cursor.lastrowid WHERE ROWID = cursor.lastrowid

3.2.3) Return cursor from 3.1.1 if cursor.rowcount > 0 else Error.

Note: autoIncrementFlag is retrieved by func. getTblNames(tbl, 1) -> mode=1 provides a tuple with autoIncrement as 3rd element in tuple.

These additional UPDATE operations would only happen in tables with autoIncrementFlag = 0, which in practice are table with compose Primary Keys.

(Link tables and other very few tables in the system).

KEEP IN MIND THIS APPROACH, as it will be most efficient and stable when writing to DB taking advantage of the autoincrement feature in SQLite.

This approach will only work with ROWID tables (it depends on the internal ROWID column). Do NOT use WITHOUT ROWID tables, as it will crash.

ALL the autoIncrementFlag buzz above can be eliminated if only ROWID is used for all tables. (fldID is no longer used in incrementing tables, as described just above).

# ----------------------------------------------------------------------------------------- #

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Version 2 of DB Writes:

Eliminates fldID in all non auto-increment tables (table with multiple PK columns, ej, Link Tables) and replaces fldID with ROWID.

IMPORTANT: CODE MODIFICATIONS ARE IN: setRecord(), setRecords(), \_get\_max\_id() (both in async\_cursor.py and krnl\_sqlite.py), dbRead and DataTable \_\_init\_\_().

All code changes are marked to make them reversible. Search for the text: "TODO(cmt): Codigo de modificacion" in all the project files.

That’s about it!!

Work to do when it's stable:

1) remove the actual fldID field from all non-autoIncrement tables in the DB.

2) rename the fldID entries for the tables in \_sys\_Fields table to ROWID (replacing the existing dbFldName)

3) In the future:

- Do NOT create a fldID field in new non-autoIncrement tables

- For ALL tables with single PK (autoincrement), make the PK field INTEGER PRIMARY KEY.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

On using ROWIDs (from the SQLite documentation):

*5. ROWIDs and the INTEGER PRIMARY KEY*

*Except for WITHOUT ROWID tables, all rows within SQLite tables have a 64-bit signed integer key that uniquely identifies the row within its table. This integer is usually called the "rowid". The rowid value can be accessed using one of the special case-independent names "rowid", "oid", or "\_rowid\_" in place of a column name. If a table contains a user defined column named "rowid", "oid" or "\_rowid\_", then that name always refers the explicitly declared column and cannot be used to retrieve the integer rowid value.*

*The rowid (and "oid" and "\_rowid\_") is omitted in WITHOUT ROWID tables. WITHOUT ROWID tables are only available in SQLite version 3.8.2 (2013-12-06) and later. A table that lacks the WITHOUT ROWID clause is called a "rowid table".*

*The data for rowid tables is stored as a B-Tree structure containing one entry for each table row, using the rowid value as the key. This means that retrieving or sorting records by rowid is fast. Searching for a record with a specific rowid, or for all records with rowids within a specified range is around twice as fast as a similar search made by specifying any other PRIMARY KEY or indexed value.*

*With one exception noted below, if a rowid table has a primary key that consists of a single column and the declared type of that column is "INTEGER" in any mixture of upper and lower case, then the column becomes an alias for the rowid. Such a column is usually referred to as an "integer primary key". A PRIMARY KEY column only becomes an integer primary key if the declared type name is exactly "INTEGER". Other integer type names like "INT" or "BIGINT" or "SHORT INTEGER" or "UNSIGNED INTEGER" causes the primary key column to behave as an ordinary table column with integer affinity and a unique index, not as an alias for the rowid.*

*The exception mentioned above is that if the declaration of a column with declared type "INTEGER" includes an "PRIMARY KEY DESC" clause, it does not become an alias for the rowid and is not classified as an integer primary key. This quirk is not by design. It is due to a bug in early versions of SQLite. But fixing the bug could result in backwards incompatibilities. Hence, the original behavior has been retained (and documented) because odd behavior in a corner case is far better than a compatibility break. This means that the following three table declarations all cause the column "x" to be an alias for the rowid (an integer primary key):*

*CREATE TABLE t(x INTEGER PRIMARY KEY ASC, y, z);*

*CREATE TABLE t(x INTEGER, y, z, PRIMARY KEY(x ASC));*

*CREATE TABLE t(x INTEGER, y, z, PRIMARY KEY(x DESC));*

*But the following declaration does not result in "x" being an alias for the rowid:*

*CREATE TABLE t(x INTEGER PRIMARY KEY DESC, y, z);*

*Rowid values may be modified using an UPDATE statement in the same way as any other column value can, either using one of the built-in aliases ("rowid", "oid" or "\_rowid\_") or by using an alias created by an integer primary key. Similarly, an INSERT statement may provide a value to use as the rowid for each row inserted. Unlike normal SQLite columns, an integer primary key or rowid column must contain integer values. Integer primary key or rowid columns are not able to hold floating point values, strings, BLOBs, or NULLs.*

*If an UPDATE statement attempts to set an integer primary key or rowid column to a NULL or blob value, or to a string or real value that cannot be losslessly converted to an integer, a "datatype mismatch" error occurs and the statement is aborted. If an INSERT statement attempts to insert a blob value, or a string or real value that cannot be losslessly converted to an integer into an integer primary key or rowid column, a "datatype mismatch" error occurs and the statement is aborted.*

*If an INSERT statement attempts to insert a NULL value into a rowid or integer primary key column, the system chooses an integer value to use as the rowid automatically. A detailed description of how this is done is provided separately.*

*The parent key of a foreign key constraint is not allowed to use the rowid. The parent key must used named columns only.*

*If no ROWID is specified on the insert, or if the specified ROWID has a value of NULL, then an appropriate ROWID is created automatically. The usual algorithm*

*is to give the newly created row a ROWID that is one larger than the largest ROWID in the table prior to the insert. If the table is initially empty, then a ROWID*

*of 1 is used. If the largest ROWID is equal to the largest possible integer (9223372036854775807) then the database engine starts picking positive candidate ROWIDs*

*at random until it finds one that is not previously used. If no unused ROWID can be found after a reasonable number of attempts, the insert operation fails with an*

*SQLITE\_FULL error. If no negative ROWID values are inserted explicitly, then automatically generated ROWID values will always be greater than zero.*

*The normal ROWID selection algorithm described above will generate monotonically increasing unique ROWIDs as long as you never use the maximum ROWID value and you*

*never delete the entry in the table with the largest ROWID. If you ever delete rows or if you ever create a row with the maximum possible ROWID, then ROWIDs*

*from previously deleted rows might be reused when creating new rows and newly created ROWIDs might not be in strictly ascending order.*

27Oct22:

Lot of work done on setRecord(), setRecords(), execute(), get\_max\_id() functions to polish all quirks. Now running good.

Quite some work done on getTblName(), getFldName() for fix and optimze them. The are fully reentrant and all checks for reentrancy are now removed.

Important: the need to check for autoIncrement is now removed. Still flags autoIncrement and isWITHOUTROWID are setup and maintanined in \_sys\_Tables (memory copy).

Important: DB modification. the dual PK is removed from all Link Tables. In order to remain compatible with autoincrement, only one INTEGER PRIMARY KEY is defined in these

tables and the required fields are set with UNIQUE constraint.

\_get\_max\_id(): modified to manage all possible exceptions. Also, a call using ROWID is always made first (since most of the tables are autoincrement-capable). Only if this fails, is a call made using db\_fldID (which is present in all tables in the system).

This method enables that the fldID column and the internal ROWID column are always kept in-sync since the data obtained from the call to MAX(ROWID) is recorded on fldID column.

**28Oct22:**

**Implementation of threaded writes with setRecords():**

The logic is as follows: When a DataTable is too big (number of records > \_\_SINGLE\_BLOCK\_SIZE, which is about 100), then a write-prioritization logic is executed that parts

the array dataList in chunks (with size \_\_THREAD\_WRT\_BLOCK\_SIZE, which is about 1000)

Then a number of threads is launched using ThreadPoolExecutor and each thread runs setRecords() with one of the chunks of that previously created. This is to balance the write

load in case very large data tables are sent for write, which could be the case during data syncrhonization

The code inside setRecords() to accomodate writing each chunks is below

# Indices p/ manejo de escritura de datos con multi-threading (para el caso de dataList muy grandes)

min\_idx = 0

max\_idx = self.dataLen # Setea indices por default para escritura de datos.

if kwargs.get('\_\_max\_idx', None) is not None:

if self.\_\_call\_from\_wrtWrapper is True:

min\_idx = kwargs['\_\_min\_idx']

max\_idx = kwargs['\_\_max\_idx']

else:

pass # si se paso max\_index, pero no se llama desde el wrapper, va a ejecutar normalmente.

Basically, uses min\_idx and max\_idx passed via kwargs to write portions of dataList.

In both cases of UPDATE or INSERT, sqlParams takes the part of dataList delimted by min\_idxand max\_idx and sends that for writing to DB.

The logic to generate the chunks (data blocks) to setup parameters for setRecords() and to launch the threads is handled via @decorator function writeScheduler()

29Oct22:

Finishing the implementation of threaded writes in setRecords for large data blocks.

a decorator is implemented: @\_\_writeWrapper that wraps setRecords() and sends to execute in multiple threads using ThreadPoolExecutor.

To do this, setRecords() code was modified to work bases in indexes: Now, min\_index, max\_index are passed by @\_\_writeWrapper, or set by default if setRecords() is not

decorated, and the logic works on the dataList structure, using the indexes to perform the DB writes.

After a good fight with parameter passing, the function is working invoqued from within the decorator, via ThreadPoolExecutor.

Important: This approach only works with AsyncCursor running. if AsyncCursor is not running, the multithreading logic is ignored.

For reference, here's the code for the decorator:

@staticmethod

def \_\_writeScheduler(func): # wrapper para lanzar setRecords() en multiple threads.

@functools.wraps(func)

def wrapper(self, \*args, \*\*kwargs):

if not isinstance(self, DataTable) or func.\_\_name\_\_ not in self.\_\_func\_names\_strings:

return f'ERR\_SYS\_Invalid function call {func.\_\_name\_\_}.'

if self.dataLen <= self.\_\_SINGLE\_BLOCK\_SIZE or self.\_writeObj.is\_stopped():

return func(self, \*args, \*\*kwargs)

# Crea threads y las lanza si dataLen > \_\_SINGLE\_BLOCK\_SIZE. Numero de threads a generar = numero de blocks.

wrtBlockSize = self.\_\_THREAD\_WRT\_BLOCK\_SIZE if \

int(self.dataLen / self.\_\_THREAD\_WRT\_BLOCK\_SIZE) <= self.\_\_MAX\_WRT\_CONCURRENT\_THREADS else \

int(self.dataLen / self.\_\_MAX\_WRT\_CONCURRENT\_THREADS)

number\_of\_blocks = int(self.dataLen / wrtBlockSize) + (1 if self.dataLen % wrtBlockSize else 0)

arg\_kwargs = []

for j in range(number\_of\_blocks):

min\_idx = j \* wrtBlockSize

max\_idx = (min\_idx + wrtBlockSize) if j < number\_of\_blocks - 1 else self.dataLen

arg\_kwargs.append({'\_\_min\_idx': min\_idx, '\_\_max\_idx': max\_idx})

retValue = []

try:

with ThreadPoolExecutor(max\_workers=number\_of\_blocks + 1) as executor:

for result in executor.map(lambda i: func(self, \*\*i), arg\_kwargs, timeout=10): # Timeout: 10 secs

retValue.extend(result)

retValue.sort()

except (Exception, TimeoutError) as e:

retValue = f'ERR\_SYS\_Thread execute failure: {e}'

return retValue

# Codigo debajo de esta linea se ejecuta una unica vez al inicializar. Son atributos de \_\_writeScheduler.

return wrapper

IMPORTANT: The threading approach will just split a big DataTable in chunks and insert those chunks CONCURRENTLY into the write queue managed by AsyncCursor.

But anything sent to the queue after that big DataTable that was split in parts will most probably end up in the queue AFTER all the objects generated by the call to

setRecords(). There may be a possibility of an insertion in the middle if a call to setRecord() is made.

With this, IT MAY BE REQUIRED TO IMPLEMENT A PRIORITY QUEUE TO BRING TO THE FRONT OF THE QUEUE ALL THE OBJECTS INSERTED BY FOREGROUND THREADS or by other time-critical

threads.. Just keep this in mind...

30Oct22:

- Added functionalities to SQLiteQuery, SqliteQueueDatabase \_\_del\_\_ destructors.

- Implemented the writeObj.stop() at the end of all the main modules in order to perform an orderly close of the databases. The join() method called inside stop() forces

the execution of both SqliteQueueDatabase AND SQLiteQuery destructors.

\*\* TODO: Define functions packTable(), unpackTable() in DataTable: these functions will convert all data the attributes to be transferred for synchronization into a Dictionary.

The, the dictionary must be converted to JSON for storage in DB.

the unpackTable() method must be a class Method, as it takes a dictionary as sole argument and returns a DataTable object.

Use list \_\_xferableItems = ('\_\_tblName', '\_\_dbTableName', '\_\_fldNames', '\_\_dbFldNames', '\_\_dataList', '\_\_operation') to determine which items are to be included in pack()

Update this list dynamically as requirements appear.

01Nov22:

- packTable(), unpackTable() already implemented and tested.

- Added wait time and a re-entry stack logic to SQLiteQuery.execute() to avoid re-entry conflicts and wait until timeout in case database is locked.

Code for SQLiteQuery.execute() is:

\_\_indicesDict = defaultdict(list) # {tbl: (max\_index, indexCounter, reentryCounter)}

def execute(self, strSQL='', params='', \*, tbl=None, fldID\_idx=None):

""" Executes strSQL. Returns a cursor object or errorCode (str).

Implements re-entry handling code to support re-entry from other threads in the time between the execution

of get\_max\_id() and the actual write to DB via the execute() command.

"""

if strSQL:

if tbl is not None and fldID\_idx is not None: # entra aqui solo en INSERT

params = list(params)

with lock: # Este lock asegura que durante reentrada 2 tbl\_idx no terminen con los mismos valores

tbl\_idx = self.\_\_indicesDict.get(tbl) # tbl\_idx = None si la tabla no existe en \_\_indicesDict

if tbl\_idx is None:

tbl\_idx = [None, 0, 0] # Inicializa estruct. tbl\_idx si tbl no existe aun en \_\_indicesDict

tbl\_idx[2] += 1 # Incrementa reentryCounter.

self.\_\_indicesDict[tbl] = tbl\_idx # Hace disponible reentryCounter para otros threads

max\_id = self.\_get\_max\_id(tbl) # Toma max index de tbl

if tbl\_idx[0] is None or tbl\_idx[0] == max\_id:

tbl\_idx[1] = tbl\_idx[2] # Si ya hay un idx en \_\_indicesDict (es reentrada), actualiza index counter

tbl\_idx[0] = max\_id

params[fldID\_idx] = tbl\_idx[0] + tbl\_idx[1] # Actualiza fldID en params.

with lock: # lock porque la actualizacion de los 3 campos de \_\_indicesDict[tbl] DEBE ser atomica.

self.\_\_indicesDict[tbl] = tbl\_idx

print(f'\*\*\*\*\*\*\*\* On entry \_\_indicesDict = {self.\_\_indicesDict[tbl]}')

while True:

with self.\_\_conn:

try:

cur = self.\_\_conn.execute(strSQL, params)

except (sqlite3.Error, sqlite3.DatabaseError) as e:

cur = f'ERR\_SQLiteQuery execute() error: {e} - {callerFunction(getCallers=True)})'

module\_logger.error(cur)

self.\_\_conn.rollback()

self.\_\_conn.execute('PRAGMA OPTIMIZE; ')

raise sqlite3.DatabaseError(f'Exception.{cur}')

finally:

break

if tbl is not None and fldID\_idx is not None:

if self.\_\_indicesDict[tbl][2] > 0:

with lock:

self.\_\_indicesDict[tbl][2] -= 1 # Decrementa reentryCounter

if self.\_\_indicesDict[tbl][2] == 0: # Si llego a cero, resetea todos los indices de tbl.

self.\_\_indicesDict[tbl] = [None, 0, 0]

print(f'\*\*\*\*\*\*\*\* On exit \_\_indicesDict = {self.\_\_indicesDict[tbl]}')

return cur

return None

*READING:*

*GIL Notes:*

*https://docs.python.org/2/faq/library.html#what-kinds-of-global-value-mutation-are-thread-safe*

*What kinds of global value mutation are thread-safe?¶*

*A global interpreter lock (GIL) is used internally to ensure that only one thread runs in the Python VM at a time. In general, Python offers to switch among threads only between bytecode instructions; how frequently it switches can be set via sys.setcheckinterval(). Each bytecode instruction and therefore all the C implementation code reached from each instruction is therefore atomic from the point of view of a Python program.*

*In theory, this means an exact accounting requires an exact understanding of the PVM bytecode implementation. In practice, it means that operations on shared variables of built-in data types (ints, lists, dicts, etc) that “look atomic” really are.*

*For example, the following operations ARE ALL ATOMIC (L, L1, L2 are lists, D, D1, D2 are dicts, x, y are objects, i, j are ints):*

*L.append(x)*

*L1.extend(L2)*

*x = L[i]*

*x = L.pop()*

*L1[i:j] = L2*

*L.sort()*

*x = y*

*x.field = y*

*D[x] = y*

*D1.update(D2)*

*D.keys()*

*These aren’t:*

*i = i+1*

*L.append(L[-1])*

*L[i] = L[j]*

*D[x] = D[x] + 1*

*Operations that replace other objects may invoke those other objects’ \_\_del\_\_() method when their reference count reaches zero, and that can affect things. This is especially true for the mass updates to dictionaries and lists. When in doubt, use a mutex!*

*Can’t we get rid of the Global Interpreter Lock?¶*

*The global interpreter lock (GIL) is often seen as a hindrance to Python’s deployment on high-end multiprocessor server machines, because a multi-threaded Python program effectively only uses one CPU, due to the insistence that (almost) all Python code can only run while the GIL is held.*

*Back in the days of Python 1.5, Greg Stein actually implemented a comprehensive patch set (the “free threading” patches) that removed the GIL and replaced it with fine-grained locking. Unfortunately, even on Windows (where locks are very efficient) this ran ordinary Python code about twice as slow as the interpreter using the GIL. On Linux the performance loss was even worse because pthread locks aren’t as efficient.*

*Since then, the idea of getting rid of the GIL has occasionally come up but nobody has found a way to deal with the expected slowdown, and users who don’t use threads would not be happy if their code ran at half the speed. Greg’s free threading patch set has not been kept up-to-date for later Python versions.*

*This doesn’t mean that you can’t make good use of Python on multi-CPU machines! You just have to be creative with dividing the work up between multiple processes rather than multiple threads. Judicious use of C extensions will also help; if you use a C extension to perform a time-consuming task, the extension can release the GIL while the thread of execution is in the C code and allow other threads to get some work done.*

*It has been suggested that the GIL should be a per-interpreter-state lock rather than truly global; interpreters then wouldn’t be able to share objects. Unfortunately, this isn’t likely to happen either. It would be a tremendous amount of work, because many object implementations currently have global state. For example, small integers and short strings are cached; these caches would have to be moved to the interpreter state. Other object types have their own free list; these free lists would have to be moved to the interpreter state. And so on.*

*And I doubt that it can even be done in finite time, because the same problem exists for 3rd party extensions. It is likely that 3rd party extensions are being written at a faster rate than you can convert them to store all their global state in the interpreter state.*

*And finally, once you have multiple interpreters not sharing any state, what have you gained over running each interpreter in a separate process?*

**28-May-23**: Looked into \_\_get\_max\_id(): The premise is that this function is called ONLY from within SqliteQueueDatabase class methods, namely, only called from the write-queue manager, so its access is serialized by the queuing code and no re-entry possible, as the code is always called from 1 thread and 1 thread only. Hence no locks are required in its code.

**31-May-23: Digression on execute() method and the use of parameters tbl, fldID\_idx:**

Tried to eliminate the use of **tbl, fldID\_idx** as parameters to be passed to execute\_sq() function, to gain in simplicity. The idea being that tbl and fldID\_idx are obtained from parameters sql and params.

Removing tbl is rather trivial using a small parsing function to extract the table name from the sql string:

@staticmethod  
def extractTblName(sql=''):  
 *"""ONLY for INSERT statements: returns the table name defined in a INSERT INTO sql statement  
 If the keywords INSERT INTO are not present, returns the original sql string."""* sqlStr = sql.strip().lower()  
 if sqlStr.\_\_contains\_\_('insert') and sqlStr.\_\_contains\_\_('into'):  
 split1 = sqlStr.strip().split('into')[1].strip().replace("'", '"')  
 split1 = split1.replace("[", '"').replace("]", '"')  
 if split1.\_\_contains\_\_('"'):  
 split1 = split1.split('"')[1].strip().upper()  
 # print(f'split1: {split1}')  
 if split1.\_\_contains\_\_('.'):  
 tbl = split1.split('.')[1].strip()  
 else:  
 tbl = split1.strip()  
 return tbl  
 else:  
 return sql # If sql doesn't contain INSERT INTO, returns the original sql string

***However***, the removal of fldID\_idx proves more difficult: fldID\_idx is needed by the execute function to assign a record index value (ROWID value) when writing in predefined blocks (\_\_sync\_db option is True). ***Synchronizing the db is a design criterion in the system***: a ROWID value must be provided by the \_\_get\_max\_id() function when \_\_sync\_db is True in order to allow for a simple way of generating unique record IDs across devices and enable the record synchronization across all devices in using the system.

A solution was found by passing a dictionary (instead of a list) as the params argument. From that dictionary, the fldID field can be derived and from it fldID\_idx. This was up and running.

*The problem is cost*, in particular with the setRecords() function: when passing fldID\_idx, the \_execute() method needs only a list as the params parameter in order to operate. The list is passed directly as follows:

cur = self.\_writeObj.execute\_sql(strSQL, self.dataList[i], tbl=self.tblName, fldID\_idx=fldID\_idx)

Here, the list is obtained directly as self.dataList[i], which is the fastest we can get

If fldID\_idx is not passed, a dictionary must be passed, and this is done as follows.

cur = self.\_writeObj.execute\_sql(strSQL, self.unpackItem(i, 1) if insert else self.dataList[i], tbl=self.tblName, fldID\_idx=fldID\_idx)

Here, a call to self.unpackIitem() (or a call to zip) must be done, significantly incrementing the overhead in my opinion, in particular when processing large amounts of records, as a dictionary must be created recursively in every run of the loop in order to turn the simple self.dataList[i] into a dictionary.

So, we will stick with tblName and fldID\_idx as parameters, to keep thing lean and as fast as possible.

**Implementation of executemany()**

executemany() can be implemented (relatively simply, I suppose), in particular for INSERT operations, using the current implementation of \_\_get\_max\_id(), and the implementation of the cursor queue.

The idea is as follows:

* executemany\_sql () is called specifically by the application code.
* In INSERT operations, executemany\_sql () calls \_\_get\_max\_id() and defines if it needs to split the write into multiple cursors. This is the case when the indices returned by \_\_get\_max\_id() exceed the MAX\_ID assigned to the device. This code is already implemented. The number of records requested to \_\_get\_max\_id() is passed in argument *row\_count.*
* executemany\_sql () accommodates all parameters and arguments, creates the necessary write cursors and puts them in the write queue.
* The writer pulls the cursors from the queue -> there must be an identifier to show that they come from executmany\_sql(). -> Done via specific call: setRecords() calls writeObj.executemany().
* The writer executes the sql string and parameters using via an \_execute() method that invokes sqlite.executemany().

executemany() returns the cursor generated by sqlite.executemany(). The application must process those results. The only valid return from executemany() is rowcount (contains the number of records successfully processed by the executemany() call).

d sqlite3.executemany() documentation: <https://docs.python.org/3/library/sqlite3.html#sqlite3.Cursor.executemany>

<https://www.geeksforgeeks.org/how-to-execute-many-sqlite-statements-in-python/>

commit important notes: <https://stackoverflow.com/questions/36243538/python-sqlite3-how-often-do-i-have-to-commit>

INSERT: <https://www.sqlite.org/faq.html#q19>

*(19) INSERT is really slow - I can only do few dozen INSERTs per second*

*Actually, SQLite will easily do 50,000 or more* [*INSERT*](https://www.sqlite.org/lang_insert.html) *statements per second on an average desktop computer. But it will only do a few dozen transactions per second. Transaction speed is limited by the rotational speed of your disk drive. A transaction normally requires two complete rotations of the disk platter, which on a 7200RPM disk drive limits you to about 60 transactions per second.*

*Transaction speed is limited by disk drive speed because (by default) SQLite actually waits until the data really is safely stored on the disk surface before the transaction is complete. That way, if you suddenly lose power or if your OS crashes, your data is still safe. For details, read about* [*atomic commit in SQLite.*](https://www.sqlite.org/atomiccommit.html)*.*

*By default, each INSERT statement is its own transaction. But if you surround multiple INSERT statements with* [*BEGIN*](https://www.sqlite.org/lang_transaction.html)*...*[*COMMIT*](https://www.sqlite.org/lang_transaction.html) *then all the inserts are grouped into a single transaction. The time needed to commit the transaction is amortized over all the enclosed insert statements and so the time per insert statement is greatly reduced.*

*Another option is to run* [*PRAGMA synchronous=OFF*](https://www.sqlite.org/pragma.html#pragma_synchronous)*. This command will cause SQLite to not wait on data to reach the disk surface, which will make write operations appear to be much faster. But if you lose power in the middle of a transaction, your database file might go corrupt.*

By default, the sqlite3 module opens transactions implicitly before a Data Modification Language (DML) statement (i.e. INSERT/UPDATE/DELETE/REPLACE), and commits transactions implicitly before a non-DML, non-query statement (i. e. anything other than SELECT or the aforementioned).

**05-Jun-2023:**

\_upload\_exempted = [‘tbl\_Upload\_To\_Server’, ] implemented. This list will hold all the tables that must not send data to server or sync with other devices.

Important because these tables ignore the index assignment logic of \_get\_max\_id (). For db efficiency reasons, all upload-exempted tables will run indices from 1 to MAX SQLITE INDEX.

**21-Jun-23.**

**Update to outerObject logic.**

*The stack model for \_\_outerAttr IS NOT SAFE*.This is because the thread-switching processed managed by the OS will interrupt threads at any time. No control over it.

Hence, there are no guarantees that, once an object is stacked up in \_\_outerAttr by an Activity method call (1st thread), the execution of the function for which it was called will finish and allow for an orderly pop of the object placed on the \_\_outerAttr stack.

Moreover, if one Activity method is interrupted by another thread, and call to *the same Activity* is performed, a new outer object will be placed on the \_\_outerAttr stack. If this second call is then switched by the OS and control returned to the initial thread, the first call on the Activity will no use the wrong outer object (placed by the 2nd thread).

If in turn this Activity method called by the 1st thread finishes, the 2nd outer object will be popped from the stack. When execution of the 2nd thread resumes, the method in that thread will use now the outer object placed by the first call on the \_\_outerAttr stack, completing a total mixup.

To solve this, a dictionary approach is used, which as a side benefit eliminates the need for the use of the dynamic\_attr() decorator.

@property  
def outerObject(self):  
 if self.\_\_outerAttr:  
 return self.\_\_outerAttr[threading.current\_thread().ident]

@outerObject.setter  
def outerObject(self, obj=None):  
 *""" Appends an object (Animal, Tag, Person) to \_\_outerAttr, which is a dict that stores 1 obj value for each thread calling the function: {thread\_id: object}. In this way threads access the correct object regardless of how and when the OS switches thread execution. This wasn't the case with the previous stack structure.* ***@param*** *obj: not None -> added to \_\_outerAttr dict. """* if obj is not None:  
 self.\_\_outerAttr[threading.current\_thread().ident] = obj

# *TODO: Is this line thread-safe?? Check..*

With this code, 1 instance of outer object is saved in the \_\_outerAttr dictionary for every separate thread that calls an Activity function, as a separate, independent key: value pair

A call to outerObject returns the outer object for the calling thread.

*This makes passing of outer objects to Activity methods totally thread-safe.*

(The logic becomes independent of the threads being switched at any time by the OS).

The one limitation is that this code is safe for re-entry within *the same thread and the same Activity class* *while using only one outer object* (that should always be the case anyway as, by design, the calls using the @property are meant to be used by one defined outer object)

It must be noted that re-entry of code within the same thread is recursion/nesting and is controlled by the code and is *always sequential*. So, recursion within one thread could be managed (if needed) with dedicated stack structure. This is not implemented for now as there is no need.

If need be, the function that needs to *run recursively and call different outer objects in those recursive calls*, will have to implement the @dynamic\_attr wrapper and turn the \_\_outerAttr attribute into a LIFO stack.

**01Sep23. Deprecation of \_\_getMaxID() function and batch assignment logic.**

As part of the implementation of the replication logic, all the \_\_getMaxID() uses and all ROWID handling in batches done in execute() / executemany() is removed and deprecated.

The low-level writing function will rely at all times on the AUTOINCREMENT feature of SQLITE, always passing **None** in INSERT operations, so as to use the next available ROWID of each table.

**25-Jun-2023:**

**Memory data synchronization**

4 parameters are being sync’d between database and memory: status (lastStatus), inventory (lastInventory), localization (lastLocalization), category (lastCategory).

The logic is that these very frequently used parameters are pulled directly from memory instead of looking them up every time in the database, in order to speed up the system.

***The objective is that every query of this parameter returns exactly the same result as if it was pulled from the database.***

This idea comes from very early on, but the solution was buggy and incomplete. Now the full treatment is devised.

To achieve this, modifications are to be introduced in the code for the specific set() and get() functions handling these parameters and the issue of code re-entry from different threads must be addressed. *These functions must be designed as multi-thread re-entrant so that the objective set above is always complied with*.

* In the case of simultaneous records being logged in DB (2 records with different field value and the *same timestamp*), the record introduced first in the database (the one with the lower fldID) will be considered valid. In short, the logic here must mirror the behavior of the MAX statement in SQL.
* *Reentry is considered only when ONE given Activity (inventory, status, etc.) is executed on THE SAME outer object (Animal, Tag, etc.) by DIFFERENT THREAD and at the same time.* The code is designed to handle this situation. The handling is done with a decorator: *memFuncReentryWrapper*.

When handling re-entry mutex locks are usually involved and 2 situations must be considered:

* Thread switching while executing the body of re-entrant code: Flags can be used as needed to signal that a re-entrant portion of the code is being executed.
* Thread switching while executing code inside the lock: In this case, the lock is already assigned so any interrupting thread will effectively be blocked and will wait until the OS returns control to the thread holding the lock, this thread releasing the lock and when the OS switches again to the blocking thread, this one resuming and acquiring its lock.

*This in fact serializes access to the shared resource* ***but*** *there are no guarantees that the requesting threads will be given the lock in the order they lined up for it*: it is up to the OS to switch to any of them in an uncontrolled order.

To solve this particular issue of synchronizing the mentioned variables with their corresponding db records, the 3rd attempt will use a dictionary of lists as follows:

**{outerObject: [t var\_value, timestamp], }**

When entering the re-entrant code, all the values are simply inserted in the list.

Re-entry from different thread is handled by inserting consecutive [timestamp, var\_value]pairs for each outerObject.

* All of the accesses to the functions write to DB, in the order they are called.
* In the very rare case that the timestamp is exactly the same for 2 threads (down to the microsecond), the value inserted first in the dictionary will prevail. This is to resemble the logic of the MAX modifier in SQL mentioned above.
* In order to implement the logic, a pair (value, timestamp) must be kept as a memory variable in order to be able to contrast the timestamp of the variable at all times, to decide on its modification.
* The functions that keep a mirror value in memory are: *\_setInventory, \_setStatus, \_setCategory, \_setLocalization.*

The result of the logic then is that when all the re-entries of code are processed, the values in lastInventory, lastCategory, lastLocalization or lastStatus correspond to the latest record (as pulled by a MAX() modifier in the fldDate column) of the corresponding db tables.

**26-Jun-2023. 4th Attempt:** The final logic attempted to cover for re-entry is the simplest: use the [var\_value, timestamp] pair from the outerObject itself to compare and update ONLY when a new value (highest eventDate) appears. The code is as follows (no wrappers needed!):

""" Chequea en todos los reentry para escribir en memoria el valor de eventDate mas alto disponible """  
if self.outerObject.lastInventory:  
 with self.\_\_lock: # lock to make the 2 lines below atomic.  
 if self.outerObject.\_lastInventory[1] < eventDate:  
 # Writes items to memory list only if data in memory is from a date EARLIER than eventDate.  
 self.outerObject.lastInventory = [eventDate, eventDate]  
else:  
 self.outerObject.lastInventory = [eventDate, eventDate] # Esta linea debiera ser atomica (Espero..)

**Future Development. 5th Attempt:** Use a simple flag in the memory structure as *self.\_\_staleFlag.*

*HOWEVER, such flag would have to be implemented in the outerObject (Animal) structure. So, it’s not efficient: basically the 2nd item in the \_lastInventory, \_lastCategory lists (which is the eventDate in each case) would be replaced by a True/False flag =🡺 Don’t make a lot of sense. Discarded.*

**27Jun23: Thread prioritization. Quite an interesting case in the use of sleep().**

The issue stems from the fact that, being all writer threads locking queues, the thread will sit idle until switched if the queue is empty. A compounding factor is that the AsyncBuffer objects that perform update/closure of programmed activities carry a significantly lower priority than the DB writing threads (the write spooler itself and the DataUploadBuffer).

The idea then came that the lower-priority threads (as the ProgActivityCursor) quit if there’s nothing in the queue, instead of blocking the thread until switch, releasing the time slot for more active/busier threads.

The Python interpreter doesn’t have a method to prioritize thread execution, due to how the GIL works (I hear). This is where the ***sleep()*** function comes in.

sleep() has a nice side feature as explained in <https://stackoverflow.com/questions/92928/time-sleep-sleeps-thread-or-process#93179>

There, it basically says:

*If you look in Modules/timemodule.c in the Python source, you’ll see that in the call to floatsleep(), the substantive part of the sleep operation is wrapped in a Py\_BEGIN\_ALLOW\_THREADS and Py\_END\_ALLOW\_THREADS block, allowing other threads to continue to execute while the current one sleeps*

Using this feature, if one puts a thread to sleep, the Interpreter will switch the thread ‘earlier’ as it detects the thread is blocked in a sleep() wait. The result is that, the longer the sleep time, the less often the thread executes, which is the final desired result.

With a few tweaks to the AsyncBuffer class, the thread\_priority was implemented.

The key changes are in the BufferWriter class where:

1. Since the queue.get() blocks the execution itself, whenever there’s a priority > 0, the fetching of the object from the queue is done with queue.get\_nowait() in order not to block the thread. If the queue is empty, an Empty exemption is handled by ‘pass’ ing the code.
2. When thread\_priority > 0, the loop() function executes a sleep() command. The number of seconds to sleep determined by the thread\_priority parameter (which ranges 0 – 10).

The idea is that threads with low priority can be instructed to execute less often, to release the computer to the front-end threads or the db write spooler. This could even be modified on the fly, using the set\_thread\_priority() method, depending on system load/situation.

The code is shown below, from the ***krnl\_async\_buffer.py*** module.

def run(self):  
 looping = None  
 while True:  
 try:  
 looping = self.loop\_func() # *TODO(cmt):the thread HALTS here: When an item is put in queue, get resumes* if self.thread\_priority:  
 # *TODO(cmt): sleeps activates thread priority: The shorter the sleep time the more often thread runs* sleep(min(self.thread\_priority \* self.\_\_BASE\_SLEEP\_TIME, self.\_\_MAX\_SLEEP\_TIME))  
 except ShutdownException:  
 krnl\_logger.info(f'{self} received shutdown request, exiting.')  
 if looping is not True:  
 self.buffer\_obj.reset()  
 return

def loop\_func(self):  
 if self.thread\_priority == 0:  
 obj = self.queue.get() # *TODO(cmt): locking get(). This line waits here until an item is available in queue* else:  
 try:  
 obj = self.queue.get\_nowait() # Cuando hay thread\_priority, no bloquea, porque se bloquea en sleep().  
 except queue.Empty:  
 return True

The initial results are quite interesting, seeing how the ProgActivityCursor gets delayed as the priority set for the thread is changed.

***If thread\_priority is set to 0, the system behaves as always****, not sleeping at all and getting blocked in the queue.get() method until an item arrives in the queue.*

The sleep time is set as

min(self.thread\_priority \*\* 3 \* self.\_\_BASE\_SLEEP\_TIME, self.\_\_MAX\_SLEEP\_TIME)

With this, if \_BASE\_SLEEP\_TIME is switch\_interval / 2 (current setting), a maximum priority of 20 will allow for 4000 thread switches between runs for the thread (8000 / 2 because of the cube in the formula above).

This is convenient for the implementation of the Data Synchronization thread, that can sleep the most time while there is no wifi or internet link to connect to server or other devices.

Tying all this together, the priorities can be set as follows (and then tweak to adjust):

* ~~DataUploadCursor (writes to DB): thread\_priority = 2 (4 thread switches between runs).~~
  + - ~~Adjustable priority based on queue.qsize~~**~~()~~ DEPRECATED as of 30Aug23.**
* ProgActivityCursor (closes ProgActivities for objects): thread\_priority = 5 (5 \*\*3 / 2 = 37.5 thread switches between runs).
  + - Adjustable priority based on queue.qsize(), already implemented.

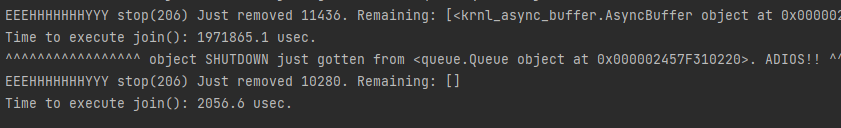
**TODO**:

* DBSyncCursor (sends/receives data through sockets to synchronize db records):
  + If no link is available thread\_priority = 20 ( 20 \*\* 3 / 2 = 4000 thread switches or 40 seconds between runs) when no connections are up.
  + If link is available:
    - Adjustable priority based on queue.qsize().

**VERY RELEVANT CONSTRAINT for Thread Priorities.**

**Threads with very low priority (long sleep times) should NOT use locks as the risk arises that if such a thread holds one lock, the rest of the system will not able to operate for as long as the thread’s sleep time. This may be code breaking for some applications.**

**08Jul23: join() can take very long to execute:**

****

*Why do it inside a lock? Probably not a good idea.*

join() executes the target thread IN FULL, effectively blocking the thread from where the call to join() is made (the *current* thread) until the execution of the target thread process is completed.

Once the target thread has finished, the **join()** method will return and the current thread can continue to execute.

Only then will the thread that made the call to join() resume its execution.

In the middle, there can (and ***there will***) be thread switches.

In normal circumstances, the main() function performs the calls to join(), so it is usually the function being suspended until the execution of the joined processes are done.

**How a deadlock happens with the buffers**.

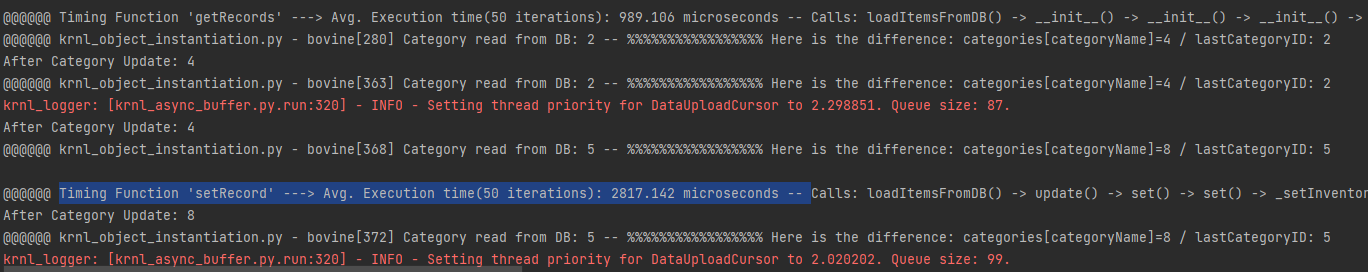
1. main() calls AsyncBuffer.close\_all\_writers() method, which in turn makes calls to AsyncBuffer.stop() method.
2. stop() calls join() for each BufferWriter thread that was created. Here, the execution flow completes the execution of each of the threads’ run() functions, which involves processing all the objects in the buffer queue.
3. The processing of these objects usually will involve writing to DB. This calls the setRecord()/setRecords() functions, which in turn access to the DataUpload buffer.
4. *Hence, if the join() method was called inside a lock block and in order to complete the processing of the items in its queue it calls setRecord()/setRecords() (which in turn go into a lock with is\_stopped() ), the processing of the queue will not be completed because of the double lock and join() will never return. A deadlock occurs.*

Solution:

* + Leave the join() statement outside any locks. No need to run it in a lock, especially for threads with priorities higher than 0.
  + Access the \_is\_stopped variable directly in the low level code, instead of calling the is\_stopped() process. This avoids the use of any locks, preventing the deadlocks.

**20Jul23:**

**Thread Interactions. Improving setRecord, setRecords performance.**

****

* As writes are done in an independent thread, it’s better to use small switchintervals, so that the write thread executes more often.
* Also, in order to speed up setRecord, setRecords, getRecords, it’s best to give the UploadBuffer a priority > 1 so that the write thread executes on every cycle and the UploadBuffer write thread doesn’t.

With this, a good balance is found leaving switchinterval = 0.005sec (default) and thread\_priority=3 for UploadBuffer*.* ***TODO****: Fine tune these values in the final version of the application.*

With this, setRecord execution time goes from 5.5 msec to 2.9 msec average.

This comes at the expense of larger queues for DataUploadCursor, which are flushed in blocks using the fast setRecords() function.

**25Aug23: As part of the move to Marmot replication, the use of UploadBuffer and \_Upload\_To\_Server table are now discontinued and deprecated.**

**31Aug23: multiple writer-objects: A Pool of writer objects, 1 object per open database (in the case of multiple databases in use).**

The code from krnl\_db\_access, class SqliteQueueDatabase that implements the pool of writer objects is below.

def \_\_new\_\_(cls, \*args, \*\*kwargs): # \_\_new\_\_ func. to create SqliteQueueDatabase objs for multiple database names.  
 *"""Creates 1 SqliteQueueDatabase object for each database passed. Stores them in \_\_instances dict, enabling  
 work with multiple databases by all threads in the system via this 'serializer' database access object.  
 After creation, always returns the same instance for each of the databases passed, as in a Singleton.  
 \_\_instances dict works as a pool of serializer-objects: 1 object for each database.  
 """* for o in cls.\_\_instances:  
 if o.dbName == args[0]: # args[0] is database argument.  
 return o # If an object with the same database name exists, returns that object.  
 o = super().\_\_new\_\_(cls)  
 cls.\_\_instances[o] = False # Sets initialized flag to False, for \_\_init\_\_() to execute the 1st time only.  
 return o  
  
def \_\_init\_\_(self, database, \*args, use\_gevent=False, autostart=False, queue\_max\_size=None, results\_timeout=None,  
 sync\_db\_across\_devices=False, \*\*kwargs):  
 # if self.\_\_initialized: # These 3 commented lines of code used with Original \_\_new\_\_() above.  
 # return  
 # self.\_\_initialized = True  
  
 if self.\_\_instances.get(self):  
 return  
 self.\_\_instances[self] = True # Sets initialized value in \_\_instances dict to avoid re-initialization.

Additionally, management of object stop() and shutdown is done in a synchronized manner with AsyncBuffer objects that write to db (a SqliteQueueDatabase writer object cannot be stopped if there are active AsyncBuffers that depend on that write object).

This is done with a dictionary of the form {database\_name: stop\_Event, } in krnl\_abstract\_base\_classes:

class AbstractAsyncBaseClass(ABC):  
 # When set, the event signals SQLiteQueueDatabase.stop() that it's ok to stop the writer.  
 buffer\_writers\_stop\_events = {} # {db\_name: Event(), }  
  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()

This dictionary is populated by the code in krnl\_async\_buffer.py:

def start(self):  
 with self.\_\_lock:  
 if not self.\_is\_stopped:  
 return False # Si el writer esta andando (\_is\_stopped=False), la llamada a start() retorna False.  
  
 def buffer\_writer():  
 # Crea el objeto writer. Pasa asyncBuffer object y queue  
 self.\_\_buffer\_writer = BufferWriter(self, self.\_write\_queue, priority=self.\_\_thread\_priority,  
 qsize\_action\_threshold=self.\_\_qsize\_threshold)  
 krnl\_logger.info(f'Starting writer object for {self.\_cursor\_type}.')  
 self.\_\_buffer\_writer.run() # Con esta llamada entra al loop de extraccion de datos del queue  
  
 # Crea thread. write\_spooler es el target.  
 self.\_writer\_thread = self.\_thread\_helper.thread(buffer\_writer)  
  
 with self.\_\_lock:  
 self.\_is\_stopped = False  
 self.\_writer\_thread.start() # *TODO(cmt): Aqui arranca el writer thread.* db\_name = self.\_cursor\_type.writes\_to\_db()  
 if db\_name:  
 # if not any(j.writes\_to\_db() == self.\_cursor\_type.writes\_to\_db() for j in self.\_\_active\_buffers):  
 # Si no existe Event en dict buffer\_writers\_stop\_events, lo crea.  
 if db\_name not in self.buffer\_writers\_stop\_events:  
 self.buffer\_writers\_stop\_events[db\_name] = Event()  
 else: # Si ya existe, resetea Event para el db-writer retornado por self.writes\_to\_db().  
 self.buffer\_writers\_stop\_events[db\_name].clear()  
 self.\_\_active\_buffers.append(self) # Internal count of all active buffers. []-> it's ok to shutdown  
 return True

and each Event’s clear()/set() modes are managed by the start() and stop() methods in krnl\_async\_buffer.py.

The code in krnl\_db\_access.py checks on the status of the Event corresponding to each object before proceeding with the SqliteQueueDatabase.stop() of an object, as follows:

def stop(self):  
 db\_logger.info('db writer environment stop requested.')  
   
 with self.\_\_lock:  
 if self.\_is\_stopped: # buffer\_writers\_stop\_events is set -> inserta SHUTDOWN en queue.  
 return False  
  
 if self.\_database in self.buffer\_writers\_stop\_events:  
 # *TODO(cmt): self.buffer\_writers\_stop\_events[self.\_database].clear() NO DEBE IR AQUI porque event esta* # *ligado SOLAMENTE al estado run/stop de AsyncBuffers y Event set()/clear() se setean desde ahi.* if not self.buffer\_writers\_stop\_events[self.\_database].is\_set():  
 # if buffer\_writers\_stop\_events NOT set there are still open buffers writing to db. Cannot shutdown.  
 db\_logger.info('Other db-accessing threads are still alive and writing data to the database.'  
 ' Please close all other threads first.')  
 # *TODO: remove this line below after testing is done.* krnl\_logger.info('Other threads are still alive and writing data to the database.'  
 ' Please terminate all other database-accessing threads first.')  
 return False  
  
 krnl\_logger.info(f'stop(): passing for {self.\_database} async write object.')  
 self.\_write\_queue.put(SHUTDOWN)  
 self.\_is\_stopped = True  
 self.\_writer.join() # join() fuerza al thread que lanzo el db writer a esperar que se procese SHUTDOWN.  
 return True

**Chapter 2: Database Synchronization**

**21Jul23:**

**The solution for synchronization will be with the use of a canned product for SQLite replication/synchronization.**

The simple way to go about this at this point is to not launch the dedicated DataUploadCursor buffer and thread, by setting auto\_start=False in *custom\_types.py.*

\_uploadBuffer = AsyncBuffer(DataUploadCursor, autostart=False, precedence=0, thread\_priority=3)

This is done now and setRecord/setRecords no longer write to \_Upload\_To\_Server table, which soon will be deprecated and eliminated from the database.

For available options on replication products see:

[**https://stackoverflow.com/questions/16032825/method-to-replicate-sqlite-database-across-multiple-servers**](https://stackoverflow.com/questions/16032825/method-to-replicate-sqlite-database-across-multiple-servers)

[**https://maxpert.github.io/marmot/**](https://maxpert.github.io/marmot/)

From the options presented, delve into Marmot.

Removing the handling of db replication/synchronization from the system, setRecord/setRecords improve to less than 3msec execution time. getRecords goes to 1 – 1.5 msec execution time.

**Problem to solve (21Jul23): detection of creation of duplicate objects:** creation of objects like Animal, Person, Tag, Device could potentially lead to duplication if 2 devices not connected create 2 independent objects with the same IDs.

The solution to this problem is defining the type of IDs to check during synchronization of devices and once a duplicate object (same IDs as per definition):

* Rename object appropriately. If needed, create a new database ID\_Object record in main Object table and remove the original records (See if this is really required).
* Update the involved registers (registerDicts) to remove replaced keys and insert new keys. Most probably the operation (at least parts of it) will have to be done under a lock.
* Assign all activities and other data linked to the objects to the new ID\_Object in the database (This should be done automatically via the CASCADE setting in the Foreign Keys for the involved records. See what happens).
* Potential duplication of Activities: When declaring duplicate objects, there’s a potential to also duplicate Activities assigned to the objects. For this:
  + In the case of regular Activities, leave all the potentially duplicated activities as they are, *without change.*
  + In the case of one-time activities (destete, castracion, etc), leave the ***first*** activity performed (earlier timestamp) and delete the activity with the latter timestamp.

This should resolve most of the conflicts due to duplication for now.

**04Jun24. Object Duplication.**

Definition: Object Duplication is the creation of separate db records in different nodes referring to the same physical object.

The potential for object duplication is inherent to the operation with multiple nodes that are not continuously synchronized (they can and will operate offline).

Duplication identification and flagging is achieved via database replication, and is an asynchronous event that depends solely on the availability and timing of the replication.

With this:

* Object duplication (the creation of separate db records in different nodes referring to the same physical object) will be identified and flagged at database level. Done in SQLite via INSERT triggers, and updated in the system via duplication and replication support functions (processReplication() and processDuplication() methods).
* The detection of duplicates and yielding the original db record to generate system objects will be handled by the getObject() method in all applicable classes.

**Duplicated objects: Criteria for flagging duplication (What to check).**

Animal: Tag IDs.

If just one of the tag IDs is used by another active animal, then there’s duplication of Animals.

Person: Person ID. The main ID of a person cannot be duplicated. Same rules as Animal duplicates apply for duplication of Activities.

Tags: Tag ID. Same rules as Animal duplicates apply for duplication of Activities.

**22Jul23: A problem previous to object duplication: OBJECT REPLICATION.**

Using Marmot or similar to replicate the database creates the obvious problem that when a replication is attempted, record numbers in the target database may well have been used already.

This is because for Marmot and the likes, *during the replication process record numbers will not necessarily be kept: This creates the issue that the DB logic CANNOT rely on using absolute record numbers to identify objects/entities.*

***Then, an auxiliary column needs to be added to tables like Animal, Caravana, Persona, Dispositivo, Proyecto, Montos. This column, named ‘UID Objecto’ in each table will the immutable like of every object to the rest of the DB tables.***

*This will be the column to be used in the Foreign Key assignments for the table as it will be immutable across all devices in the system, regardless of the rowid it is assigned in the table.*

*This column* ***cannot*** *be generated by sqlite as it needs to remain unchanged when the ROWID of the row changes. Its value will have to be defined by the python code.*

To implement this column probably using the uuid module in Python is the way to go, both for uniqueness and simplicity.

The model, as of now, is peer-to-peer database structure, which means there will NOT be a main server and all the database files will be replicated and identical across all devices.

Now, devices change as phone sets, tablets, etc. are eventually replaced. Whenever a new device comes online, it will receive a current copy of the system database (the same existing in all active devices). Then, it would be sensible that the device, if anything (and not the database instance since they all become identical after duplication), uses an identifier to create Animal, Person, Tag objects, just to provide a tracking chance of where those objects were created, although it’s not strictly necessary.

In fact, the first try will be to just use a randomly generated UUID, without any kind of device or database identification, as this should be the most efficient solution.

**Duplication. Duplicated objects.**

With this new logic, the problem of independent generation of duplicate objects will be handled during the replication process and through a periodic routine run in the background that checks for duplicates and when duplicates are detected, the system will choose one of the Object UIDs generated and will propagate it to the rest of the devices with duplicate objects, re-writing the ‘Object UID’ field in the corresponding tables.

The UID of the first object created (chronologically speaking) is the one that will remain. All the rest are overwritten.

Listo.

UIDs implemented in: tblAnimales, tblTMMontos, tblDispositivos, tblCaravanas, tblPersonas, tblProyectos,

IMPORTANT NOTE 01Dec23: Other types of duplication, in particular multiples entries for the same event (Activity) by multiple devices is NOT supported and may NEVER be supported as independent recording of events by different devices is not always possible to determine as being the same event or not.

**In short, different users should be careful not to record the same event on separate devices as event duplication will result.**

UID Management function run during initialization and by background Duplication-management funcs (this one is from krnl\_bovine.py. Similar ones are in Caprine, Geo, Tags and all classes that require uids to operate).

@classmethod  
def \_init\_uid\_dicts(cls):  
 *""" \*\*\*\*\* This method MUST support multiple access. In particular, WILL NOT remove items from dicts \*\*\*\*\*  
 Initializes uid dicts upon system start up and when the dict\_update flag is set for the class by SQLite.  
 This method is run in the \_\_init\_subclass\_\_() routine in EntityObject class during system start, and by  
 \_processDuplicates() run by background functions.  
 Uses a checksum of \_\_active\_uids\_dict.values() to determine if there are changes to the dict.* ***@return****: None  
 """* sql = f'SELECT \* from "{cls.tblObjDBName()}" WHERE ("{getFldName(cls.tblObjName(), "fldDateExit")}" IS NULL '\  
 f'OR "{getFldName(cls.tblObjName(), "fldDateExit")}" == 0) ' \  
 f'AND "{getFldName(cls.tblObjName(), "fldFK\_ClaseDeAnimal")}" == {cls.animalClassID()}; '  
 temp = dbRead(cls.tblObjName(), sql)  
 if not isinstance(temp, DataTable):  
 val = f"ERR\_DBAccess cannot read from table {cls.tblObjDBName()}. Internal uid\_dicts no initialized. " \  
 f"System cannot operate."  
 krnl\_logger.warning(val)  
 raise DBAccessError(val)  
 if temp:  
 idx\_uid = temp.getFldIndex("fldObjectUID")  
 col\_uid = [j[idx\_uid] for j in temp.dataList] # temp.getCol("fldObjectUID")  
 idx\_dupl = temp.getFldIndex("fld\_Duplication\_Index")  
 col\_duplication\_index = [j[idx\_dupl] for j in temp.dataList] # temp.getCol("fld\_Duplication\_Index")  
 dicto1 = dict(zip(col\_uid, col\_duplication\_index)) # Reference dict with all values  
  
 # Check for differences in dict values. Must update \_\_active\_uids\_dict, \_\_active\_duplication\_index\_dict.  
 try:  
 """ There is a long-shot chance that the checksum below fails when 2 or more items change and the   
 # resulting sum remains unaltered. Then, there's an efficiency-driven bet that such a scenario will   
 # NEVER materialize. IF it ever does, it will correct itself next time the checksum changes, and   
 # all db records will be properly updated and re-assigned. """  
 checksum = sum([UUID(j).int for j in col\_duplication\_index if isinstance(j, str)])  
 except(TypeError, ValueError):  
 checksum = cls.\_\_duplic\_index\_checksum # On checksum failure, must exit with checksums unchanged.  
 temp\_dict = {}  
 if checksum != cls.\_\_duplic\_index\_checksum:  
 # Initializes \_\_active\_Duplication\_Index\_dict ONLY FOR DUPLICATE uids.  
 # An EMPTY \_\_active\_duplication\_index\_dict means there's NO duplicates for that uid in the db table.  
 cls.\_\_duplic\_index\_checksum = checksum  
 dicto2 = {k: v for k, v in dicto1.items() if v is not None} # dict with None values stripped off.  
 # for item in col\_duplication\_index: # item is a \_Duplication\_Index value.  
 duplic\_values = list(dicto2.values())  
 for item in duplic\_values:  
 if item is not None and duplic\_values.count(item) > 1:  
 # If item appears more than once in col\_duplication\_index, it's a duplicate item.  
 # Gets all the uids associated to \_Duplication\_Index for item.  
 uid\_list = [col\_uid[j] for j, val in enumerate(col\_duplication\_index) if val == item]  
 # ONLY DUPLICATE ITEMS HERE (\_Duplication\_Index count > 1), to make search more efficient.  
 temp\_dict[item] = tuple(uid\_list)  
  
 if temp\_dict: # There are changes in \_Duplication\_Index values. Must update both dicts.  
 with cls.\_\_lock: # Both must be updated in the lock block.  
 cls.\_\_active\_uids\_dict = dicto1  
 cls.\_\_active\_duplication\_index\_dict = temp\_dict  
 else:  
 # If there's differences in dict keys only -> Initializes / updates \_\_active\_uids\_dict only.  
 if set(dicto1) != set(cls.\_\_active\_uids\_dict.keys()): # Changes in keys only. Updates \_\_active\_uids\_dict  
 cls.\_\_active\_uids\_dict = dicto1 # THIS IS A SHARED DICT. The assignment operation should be atomic.  
 return None

**24Jul23. UIDs:**

*UIDs are generated via uuid4(). ALL UIDs are converted to hex and stored and processed as str (TEXT in sqlite). As follows:*

fldObjectUID=str(uuid4().hex)

With these UIDs, object IDs become independent of ROWID and the replication with other peer databases can be done effortlessly (almost).

IMPORTANT NOTE:

* fldObjectUID store a SOLE item, type TEXT in the sqlite database file. On reading it, sqlite returns a text string that results in a well formed uuid4 hex string. All UID strings stored in regular TEXT columns will NOT show any quotes.
* With JSON is different: since the column in the database file is JSON and not TEXT, UID values need to be converted to string before storage. Then all UIDs stored in a JSON column will require quotes "" so that the JSON converters adapters and decode the data properly as text, and not confuse it with bytes.

***TODO:*** *Remains to solve (as mentioned above) the issue of duplication of objects when 2 or more devices create objects referring to the same physical object (Animal, Person, etc.).*

**29Jul23. Checking for table updates. Checking for duplicate objects.**

A number of tables and table information is read into memory and data structures are kept in memory related to those tables. Examples: Animales, Personas, Caravanas, Geo Entidades, etc.

With database replication, the problem arises that:

1. *as tables are updated by an external agent (Marmot or similar) the table contents may vary and the associated memory data structures become outdated or out-of-sync.*
2. *A subproduct of this situation is that a unique object (Animal, Tag, etc.), may be defined and instantiated more than once if several devices, not connected among them process the same object before the databases have the chance to fully replicate.*

*The 2 issues are unavoidable: a) requires a dedicated solution as part of system integrity.*

*Once the solution for a) is defined, b) is addressed with specific logic to detect duplicate objects and amend database tables as required.*

As the replication/update of database tables is in essence asynchronous, the solution will be implemented via a dedicated buffer (with its dedicated thread) that will check for flags (cursor) defined for the tables that need to be monitored:

As new records are inserted (or existing records are updated) by the replicator, cursors will be enqueued so that the buffer functions that execute periodically will find items in the queue and will process them in the order received.

This has several advantages:

* Virtually immediate action when a change in a table is detected (the function is run very shortly after a cursor is queued).
* Very little/no overhead to run the test logic (same already implemented with existing buffers) that can run in period of milliseconds or seconds, as required.
* Once a change on a table occurs, the first code that will run for it is the code in these functions, which can update all the memory data structures associated with it, and then make those changes immediately available to all the threads running in the application.

This is a major advantage, as the update code for the memory data structures is run ONLY when changes are detected, making it an efficient approach.

**Ideas:**

Using the SQLite Tracing API is the most efficient and preferred way to intercept INSERT in the required tables.

***INSERT is for the moment the only operation that needs to be intercepted. UPDATEs and DELETEs have no effect on the conditions that we’re trying to cover.***

If a sqlite trace callback function gets complicated, could implement internal flags in the Python code (one for each table) that tracks the timestamp of the last INSERT for the table. Once a value higher than the value stored is detected, the logic is run to identify and processes the added/modified/deleted row. se

* Could be an identifier that’s unique to the database (a code or similar related to the database name): something that allows the Python application to univocally detect a record that is coming from a different database.

All the records in the selected tables are created with a default value code in the fldBitmask field. Once a record is by the monitoring code as not belonging to the local database, it is processed and all the applicable data structures are updated. Once all is done, the fldBitmask value is updated to the local database code, so that the record is not processed again in future calls of the monitoring function.

This will require to perform a getRecords() of all the table in every cycle of the function, but may be a price to pay if the trace callbacks cannot be implemented.

**MOST OF THE REMAININIG OF THIS CHAPTER IS DEPRECATED.**

**Left here because some concepts are still in use.**

**03Aug23. Resolution:**

To resolve the above conflict, a monitoring routine will be run (from a AsyncBuffer for now) that will load all the objects with no exit (exitYN=NULL) from each of the tables to be monitored and will run a comparison logic to detect repeat objects and records.

When a repeat object is found, the database record found will be UPDATED with the UUID corresponding to the Original object. This assures that any records created for the Repeat object are re-directed (cascaded) to the Original object as per the UPDATE rules in the DB file.

Some guidelines (03Aug23). Coming straight from comments recorded on the phone.

* Una lista de clases: [Animal, Geo, Tag, Person, Device]

desde la que la funcion de Buffer periódica ejecutara el monitoreo.

Para cada clase, 3 classmethods:

table\_changed(); verifica si hubo cambios en la tabla de objetos que tengan que disparar la actualuzacion de estructuras.

resolve\_duplicates(): cuando hay un table\_change, verifica y elimina posibles objetos duplicados (los generados independientemente por 2xo mas dispositivos que no tuvieron oportunidad de sincronizar sus databases.

update\_struct(): actualiza las estructuras \_\_registerDict y demas estruct. decdatos en memoria cuando table\_change() retorna True.

Cada clase define 2 class atrributes:

local\_rec\_counter (int): +1 cuando se inserta objeto nuevo.

-1 cuando se elimina objeto o cuando se pasa objeto a Inactivo.

Se resetea a 0 en cada ejecución de table\_changed()

checksum: SUMA de los UUID (convertidos a int) de TODOS los objetos activos (isActive = True)

Estas 2 variables son actualizadas por funcion table\_changed().

La logica de table\_changed() es:

Se hace checksum dectodos los activos leyendo la tabla completa de db.

Si el check sum nuevo es igual al almacenado retorna False ( indica no hace falta correr update\_struct)

Si checksum cambió:

- si la cantidad de records leídos es igual a la cantidad de records de antes + local\_record\_counter: retorna False (no actualizar nada porque los cambios a la tabla fueron hechos localmente por la aplicación)

(Otra class variable: **table\_record\_count**, a ser actualizada en cada llamada a la función).

- si la cuenta de records es DISTINTA de la suma total\_record\_count + local\_record\_counter, entonces se insertaron records nuevos que vienen del replicador: retorna True.

En todos los casos: actualizan checksum, total\_record\_count con los ultimos valores leídos de d y resetea a 0 local\_rec\_counter.

local\_record\_count se incrementa en las funciones register() y se decrementa en las funciones unregister()

La funcion update\_struct() se podrá ejecutar inmediatamente cuando aplique, o hacer enqueue() de un cursor. Ver qué conviene.

El codigo de monitoreo (funcion Buffer) debiera ejecutarse en el orden de minutos: cada 1 minuto aprox: la lógica es que el agregado de objetos nuevos será más o menos frecuentes pero cada vez que hay un agredado los cambios en db se deben propagar rapidamente a las estructuras de memoria.

La fn de monitoreo toma la lista de clases inicial e invoca los 3 metodos definidos arriba para cada una.

Repetición de Objetos.

Para detectar objetos repetidos (declarados múltiples veces via distintos dispositivos) hay que agregar campo timestsmp en las tablas a monitorerae (Animales, Caravanas, etc).

La deteccion y manejo de repetidos es una operacion TOTALMENTE LOCAL a cada dispositivo y db.

Incluye: ejecutar código para encontrar repetidos.

Flaggear registros Original y Repetido en db.

Eliminar repetidos de estructuras en memoria y agregar (si hiciera falta) el original en las estructuras de memoria.

Registros repetidos:

NO se pueden borrar registros que se detecten repetidos.

Esto es debido a que en el proceso de duplicacion se prowpagará el borrado a todas las db, y al borrar el record en la db que creó el repetido podrian quedar multiples registros de Actividades y demas huerfanos.

Es por eso que los repetidos:

1. Se les hace un UPDATE de su UUID (para que esa modificacion se pripague a todos lo records asociados ya existentes).

2. Se les setea el flag Repetido, para que no se los cuente como objeto Activo ni se los cargue en memoria --> Se podría usar entonces de Flag Activo para manejar las repeticiones, (en vez de crear un nuevo flag Repetido).

MEJOR AUN:

Se le da BAJA (Salida) al record repetido (se le asigna un Exit Date) y se define una categoría nueva de Baja Por Repetición.

Este esxun tipo más de Baja que se procesa en la función BajaAnimales.

Como parte de las modificaciones de las estructuras en memoria en el manejo de repetidos en el caso de los \_\_registerDict:

1. Cambiar (actualizar) el key del dict.

2. Actualizar el UUID del objeto (usar metodo setID ) en el mismo dict.

Con esto, no hace falta tocar más nada (en principio)

**Update:**

**checksum and local\_register\_counter are deprecated and replaced by *full\_uid\_list* and *new\_local\_uids*.**

They work as follows:

* When the periodic routine is run, a full list of all UIDs is pulled and stored in full\_uid\_list (using method getCol()).
* When a new object is created, the register() function will add the record UID to the new\_local\_uids.

With these 2 variables set for each class (Bovine, Caprine, Tag, Person, etc) by the periodic monitoring function, the logic will work as follows:

* + Full list of UIDs is pulled for the table and a comparison using sets is performed:
    - If all items in the just-pulled UID list is found either in *full\_uid­\_list*  or in *new\_local\_uids* list, it means that the new records created were created locally by the application, then there’s no need to perform the duplicates verifications.
    - If a record is found in the just-pulled UID list that is not in the 2 mentioned above, it means the record was duplicated from another device/database, and the logic to detect duplicates is run.
    - Once the steps above are completed, *new\_local\_uids* is reset to [] and *full\_uid\_list* is set to the just-pulled UID list.
    - The function ends.

**04Aug23. The problem of object duplication (tuplication, repeat objects)**

Design requirements for the system are:

* It must run as a multi-peer environment, with no server.
* Database replication will be, at best, eventual. That means that all records in all databases that comprise the system ***will eventually*** be fully replicated. When, how long that takes to happen, and for how long the full replication can last will never be known.
* Each of the nodes (devices) must be able to fully operate offline, and for long periods of time, if required.

With these constraints the problem arises of 2 or more nodes (devices) creating each one independent objects (Bovine, Caprine, Tag, Person, etc.) that refer with the *same physical object*. ***This is when object duplication happens***.

An extension of the problem immediately shows that more than 2 nodes can create more than 2 separate objects for the same item. That’s object tuplication.

*It must be noted that the odds of duplication happening are extremely low, however the tools to manage tuplication must be embedded in the system*.

Options to manage tuplication.

1. Leave it alone: allow for eventual repeat objects to coexist in the system, as they are created and execute detection/management only in the operations and functions that require ‘repeat-awareness’ (example: object count, object exit/removal, object location, health/sanitation historical records, etc.).

The simple implementation contrasts with the complexity required to manage operations that need to be aware of the duplications.

1. Run routines to detect repeat objects and eliminate the repeats, leaving only the original object. The problem with this approach is that it is impossible to know when the conditions to detect repeat objects are met, since the detection of repeats must be based in the checking of features and ID numbers associated to the object, which in turn are stored in database records that are subject to replication. Hence it is not known when such ‘ID records’ will be made available in each node in order to perform a successful check of repeat objects.

This approach allows for an effective way to pass around ‘ID items’ that uniquely identify each object (Tags, ID numbers, etc) *along with the creation record*:

Create a *list or a dictionary* as 1 field in the Object table (tblAnimales, tblCaravanas, etc.) that holds all the ID items for the object:

{id\_number1: id\_type1, id\_number2, id\_type2, }

This way, when a record is created, the data necessary to ID the object is included in the record created the need for additional replication of records is removed.

*The counterpart complexity is that the system must implement the logic to maintain this field up-to-date when the identification items of the object change (additions, removals).*

Some advantages:

* + Simple definition of the “original” or initial Object created: it is the object with the *earliest fldTimeStamp* record. As new repeats are found, the “original” changes based on this criterion, making all ‘later’ instances repeats.
  + A key one: As the UID fields are updated in the repeats (replaced with the UID of the Original object), the changes are cascaded to all the linked records using the Foreign Key action feature of SQLite (On UPDATE, CASCADE), making this process the most efficient possible. This can be executed multiple times for a record, as the Original and Repeats change, and is completely transparent to the applications.
  + When adding/removing ID items on an Object record, these changes are transparently propagated via replication to all nodes, making the updated ID items lists available for use by all the db functions.
  + To manage Repeats, when a Repeat is found, it is ‘taken offline’ by executing a “Baja-Repeat” method, and the exit date for it is set as current datetime. From there on, that record is no longer considered in the \_\_registerDicts and only the Original record (the one with *fldExitYN=NULL*) is loaded and used.

1. sdfsdf

**05Aug23. Enabling & managing replication of Object tables**.

In order to orderly manage the replication of Object tables (Animales, Caravanas, Personas, Geo, Dispositivos), the design is:

Updates to the memory structures must be executed by ***monitoring code*** when:

* 1. 1 row is added (INSERT statement). This will be propagated to the rest of the nodes as an additional row that is inserted to the respective table in all nodes. This will trigger an update in \_\_registerDict, plus any other updates required.
  2. 1 existing row is updated (UPDATE statement). This is propagated as an update to an existing row so detection of the updated row must be devised and actions to be executed must be specific for the modified field(s) in the record.
  3. 1 existing row is DELETEd: this will be implemented in a later stage and will essentially will trigger a Baja for the row being deleted.

1. INSERTs. Sdfsdf

In order to detect external inserts the application keeps a list of the local inserts for each object class in variable *new\_local\_uids. -> INSTEAD OF uids, this must be a list of fldID.*

1. UPDATEs.

**Design criteria**: UPDATEs should be used only for UUID field, that is, to modify the UUID field in the case of repeat objects. As far as here, this field (UID\_Objeto) and fldIdentfiers are the only updates that should be monitored as they impact the links to other records (Activities, ProgActivities, etc). UPDATES of any other fields of tables Animales, Caravanas, etc, are of no effect to the linked Foreign Keys and therefore can be UPDATEd and replicated silently by Marmot, with no additional actions required.

The problem must then be resolved for the following fields (at the time of this writing):

* + - **FldObjectUID** (to update \_\_registerDict keys and object ID)
    - **FldIdentifiers** (to update \_\_identifiers, \_\_tags, \_\_personID, etc.)
    - **fldExitDate** (to remove object from \_\_registerDict if not the Original obj.)

using 1 trigger that checks for changes in each of these 3 fields and calls the callback function whenever a change in values is detected, passing as arguments fldName and fldValue to the callback function. The use of an external function is described below.

In order to manage the updates of existing records generated by other nodes, the following is implemented:

* + 1. Define the following fields in every Object table:

*FldUPDATE: TEXT. Stores the* ***OLD*** *UUID of the record if there was a UUID change.*

The application code will keep a dictionary of the form {fldID: fldUPDATE} and each run of the periodic code will check the newly read data from the Object table with the stored dictionary. When a change is detected in one record for *fldUPDATE* field, it means that the field has been updated and the code to update the data structures associated with that field must be updated.

*fldIdentifiers:* JSON List. A list of all the identifiers assigned to the object in order to uniquely single it out in the system.

Additionally, *fldExitDate* is already defined and is another field that the trigger must monitor for changes.

* + 1. SQLite Trigger:

AFTER UPDATE

BEGIN

# create a dictionary here {i: OLD.i, }, using

create\_function(*name*, *num\_params*, *func*). See <https://docs.python.org/2/library/sqlite3.html#sqlite3.Connection.create_function>

<https://stackoverflow.com/questions/33021255/launch-a-python-script-from-a-sqlite3-trigger>

“*If the "UPDATE OF column-name" syntax is used, then the trigger will only fire if column-name appears on the left-hand side of one of the terms in the SET clause of the* [*UPDATE*](https://www.sqlite.org/lang_update.html) *statement. “*

Can use multiple triggers, 1 for each column name that needs to be monitored

* ~~1 TRIGGER for each Column that needs monitoring.~~
* ~~To define, how to “append” values from triggers fired sequentially to fldUPDATE, so not data is lot for all the fields that may change in an UPDATE (Can be more than 1).~~ *~~-> Do this with a callback function that:~~* 
  + - ~~Processes the DB field name to the internal key field name.~~
    - ~~When new value is different from old value takes the existing dict stored in fldUPDATE and updates the dict with the new key field name: fldValue pair.~~

~~Keep in mind when comparing dicts:~~

*~~d1 = {'f1': 1, 'f2': 2}~~*

*~~d2 = {'f2': 2, 'f1': 1}~~*

*~~d2~~**~~== d1 -> True~~*

~~Meaning that as long as the keys and value are kept identical, the order of the dict items is irrelevant, which is exactly what’s needed to compare the dict in fldUPDATE against the immediately previous version of itself.~~

**08Aug23. processReplicated() function.**

processReplicated() is a class function that is to be run as an IntervalFunction, with 10 second period. It inspects ALL the records in the object tables (Animales, Caravanas, Personas, Geo) and processes ***new records, updated records and repeat records*** in these tables (repeats occur when 2 or more nodes are not connected and process independently the same object. “Same object” is detected by inspecting the object’s assigned Identifiers).

What it does:

1. Loads the full object table from DB.
2. Checks for “external” records (new records INSERTed by other nodes) by comparing loaded fldID values with memory-stored variables.
   1. When external records are found, create the necessary objects and updates the \_\_registerDict.
   2. Updates all internal variables used to detect external records.
3. Checks for duplicated/repeated records: when a record in the table is found to have \_\_identifiers matching an existing object, it’s deemed as a repeat object.
   1. The logic for repeats asserts the object with the oldest (earliest) timestamp field as the Original and all the other objects sharing at least 1 of the identifiers with the Original as Repeat objects.
   2. As part of the process, the Repeat objects are left in the database, but exitDate is set for them, and they are removed from \_\_registerDict so that all subsequent operations are address via the Original object exclusively.
   3. Updates all variables used to manage repeats.
4. Checks for UPDATEd records: any record in the table that may have been updated by another node.

Operations 2, 3, 4 are completely asynchronous and can be executed by any of the nodes in the system (while online or offline) and at any time. Additionally, as the replication is performed by an external utility (Marmot) there is no possible means to “trigger” the actions in order to capture the changes.

This is the reason a periodic function has to be used (run every so many seconds) in order to scan for those changes and process them accordingly

~~Given the number of tables (less than 10) and the fact that the operations mentioned above are extremely infrequent, the functions~~ *~~will be executed in a circular queue, one at a time,~~* ~~as the monitoring function is run~~. -> This “circular queue” logic is replaced by a dedicated Trigger\_Tables table in DB and an interval function that checks on changes on that table and fires the corresponding *processReplicated()* when changes in a table are detected. See below notes of 15Aug23.

**09Aug23. Related to UPDATEs: Using fldUPDATE as a counter flag.**

See implementing one trigger for each table that modifies the fldUPDATE field for ANY update of ANY of the fields in the row, and then processReplicated() loads the full record in memory again. This is the only way to ensure any and all changes to a record is reflected in the memory structures.

The flag in fldUPDATE is a simple incremental monotonic counter that increments in 1 everytime there is an UPDATE to the record. This counter is replicated across all nodes and is modified by any of the nodes on UPDATE. The counter is NEVER reset and then ensures that all changes are replicated and eventually reach all of the nodes.

As a side advantage, also simplifies the logic for triggers: Whenever the counter in fldUPDATE read from DB differs from the value in memory, the memory object attached to the record must be completely updated, and any other structures that may require action (\_\_registerDict, myTags, etc).

*2 Fields will require special consideration: fldObjectUID, fldExitDate. (fldIdentifiers is updated transparently by updateAttributes() method).*

But the specific treatment of these fields is done in the python application, avoiding trigger complexity in sqlite.

The simplest and most efficient way to go around updating memory objects is to drop the existing object and replace it by a new object in memory with the updated data read from the db record.

**10Aug23.**

Implemented *updateAttributes() method for Animal (must implement for Person, Tag, etc). Invoked by processReplicated() in order to UPDATE any fields that are found modified from another node.*

~~Circular queues to execute processReplicated() in the background: Implement with a dqueue object and its~~ *~~rotate()~~* ~~method. Simplest, most efficient way to do it.~~

~~Just rotate() every time and execute the 1~~~~st~~ ~~object in the dqueue, in the lines of:~~

~~c\_queue = dQueue()~~

~~if hasattr(cls, ‘processReplicated’):~~

~~cQueue.append(cls)…. # Adds classes that have processReplicated implemented.~~

~~cQueue.rotate()~~

~~cQueue[0].processReplicated()~~

All deques have been deprecated now and replaced by a logic using \_sys\_Trigger\_Tables table and DB\_ID fields in selected tables in order to execute the code required for modified tables.

**22Aug23. processReplicated(): DELETE of Records. Baja of Animales.**

Need to add logic for deletions to processReplicated function.

As a general concept, for all these replication management functions, there should be logic that handles all 3 db operations: INSERT, UPDATE and DELETE.

INSERT/UPDATE are already taken care of in processReplicated. Now the description for DELETE.

In the case of Animals: 2 events must be handled:

1. **DELETEd records**: Any deletion of records from tables Animales, Caravanas, etc are an exception and here it will be assumed that the node performing the deletion will have executed all the cleanup required prior to the DELETE of the record. This include running the baja() method, etc.

When a record is deleted, all that is required in processReplicated is to remove the object (Animal in this case) from \_\_registerDict (if it’s still there).

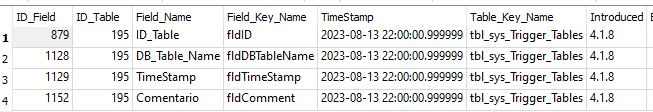
1. **Exit of an Animal via baja() method:** In a similar way as above, when an Animal is removed via baja() method, all the cleanup and closing of associated records is done by the node executing the baja() method. These records will eventually be replicated to all nodes, so the only thing left to do for in processReplicated is remove the object (Animal in this case) from \_\_registerDict (if it’s still there).

**13Aug23. Election Day. A better solution for the handling of replicated data.**

Situation:

* All the replication management, including resolution of eventual conflicts in replication is left to the replication application (Marmot).
* The system code must address the management of memory structures tied to database data that may become inaccurate/outdated due to the replication process.
* Only tables that are associated to memory structures (that need updates) will have triggers.

With this:

* 1 dedicated sys table (\_sys\_Trigger\_Tables) will be created with the following structure:
* 

Table\_Name (dbTableName). TEXT: Name of the modified table.

TimeStamp: TIMESTAMP: DateTime of the modification.

* A Trigger\_Table is a DB Table that requires the use of triggers to “notify” the python application that changes have been made and hence the memory structures need to be updated.
* “Modification” means INSERT, UPDATE or DELETE.
* 3 Triggers defined for each of the Trigger:
  + 1 “general” Trigger that fires for every INSERT and DELETE of a Trigger Table.
  + 1 “update” Trigger that fires for UPDATEs of specific fields and updates the specific row that was updated in the Trigger Table (already implemented via the fldUPDATE field, with associated logic). This trigger performs 2 actions:
    - Updates the row fldUPDATE field.
    - Updates the table row in \_sys\_Trigger\_Tables table.
* With these triggers, the logic will work as follows:

1. The database triggers update \_sys\_Trigger\_Tables in each node and the specific rows in each table as the INSERT, UPDATE, DELETE operations occur.
2. The replicator (Marmot) replicates all tables, including \_sys\_Trigger\_Tables to all nodes in the system.
3. The “system” (the python application) runs an IntervalTimer function every 30 seconds or so that only reads the \_sys\_Trigger\_Tables and detects changes.
4. For every table that is found with changes, enqueues a cursor in a dedicated AsyncBuffer, with the respective function to execute the code defined for the table been modified.
5. AsyncBuffer logic processes (in a separate, independent thread) the cursors and the functions run for each cursor update the relevant memory data structures.
6. In the case of UPDATEs, the design concept behind the updateAttributes() method in all classes is that at all times the values in the memory structures (objects, lists, dicts) reflects the values of the database rows they are associated with. This logic must be respected in order to maintain integrity with the workings of the replication application.

These triggers are basically a flag for each table indicating: the Table has been modified in 1 or more of its records, by anyone of INSERT, UPDATE, DELETE. Take action on these modifications as applicable..

Advantages of this approach:

* Fast, efficient check for table changes: Since a small, fixed-size table is read from db (\_sys\_Trigger\_Tables will usually carry less than 20 records), with cursors created and enqueued for separate processing, the function executes fast and can be run more frequently (every 30 sec).
* The AsyncBuffer logic includes an efficient processing algorithm that verifies queue activity and thresholds can be set to execute more often when a number of items are found in the queue. With this, the logic can idle when no items are in the queue, and as soon as items are present in the queue, they can be prioritized and execute immediately, so as to speed-up the update of memory structures.

**Update of 25Aug23. FldUPDATE\_counter -> Replaced by UUID.**

fldUPDATE\_counter was initially designed as an incremental monotonic counter implemented at row level to detect UPDATEs on the row: In each update, the table UPDATE\_Trigger would increment fldUPDATE in 1, and that value would be stored in the table class in fldUPDATE\_counter attribute.

f'CREATE TRIGGER IF NOT EXISTS "Trigger\_{dbTblName}\_UPDATE" AFTER UPDATE {col\_list\_str}' \  
 f' ON "{dbTblName}" FOR EACH ROW BEGIN ' + \  
 (f' UPDATE "{dbTblName}" SET "Record UPDATE" = IFNULL(old."Record UPDATE",0) + 1 WHERE ROWID=new.ROWID;'  
 if "Record UPDATE" in fldNames.values() else '') + \  
 f' UPDATE \_sys\_Trigger\_Tables SET TimeStamp=DATETIME("now","localtime") WHERE DB\_Table\_Name="{dbTblName}";' \  
 + f' END; '

Now, the problem arises when 2 nodes perform 2 independent UPDATEs on the same row of the same table.

The logic is that if the replication of the 1st update doesn’t reach all nodes in time, the replication of the 2nd update (latest in time) is the one that must reach all nodes eventually.

But for the nodes that were reached by the first update, if the value of fldUPDATE field is the same for the 2 nodes, the 2nd UPDATE will not be executed on the tables that were reached by the 1st UPDATE.

So: UUIDs will be implemented in field fldUPDATE in the UPDATE Trigger, replacing the simple counter with a counter that starts from an OFFSET. That offset is a python-generated random integer. Then, with a unique starting value incremented by the Trigger every time, the problem of duplicate values is resolved and the last (latest) UPDATE performed on a row will eventually reach all nodes.

The below Trigger line in UPDATE\_Trigger

f' UPDATE "{dbTblName}" SET "Record UPDATE" = IFNULL(old."Record UPDATE",0) + 1 WHERE ROWID=new.ROWID;'

is modified as follows:

f' UPDATE "{dbTblName}" SET "Record UPDATE" = IFNULL(old."Record UPDATE", {randrange(10000, 15000)}) + 1 WHERE ROWID=new.ROWID;

The rest of the trigger is unchanged.

The last value of the fldUPDATE field with the individual counts for each row of the table is kept as a class attribute in \_*object\_fldUPDATE\_dict {}*

The solution is fully consistent as long as the triggers in the database are not deleted. In the event of deleting a trigger and re-creating it upon system startup, a new random offset number is generated, with extremely low probability of repeating one of the existing numbers, so we’re good.

**15Aug23:**

As of today, 2 structures will be kept in memory, and updated as described above using SQLite Triggers, AsyncBuffers and IntervalTimer functions:

* + \_\_registerDict for the many objects (already implemented)
  + myTags (Animals only)

The following structures currently read from memory, will be read exclusively from Database to avoid contorted updates (until further testing to see what’s convenient):

* + lastLocaliztion (memory data)
  + lastInventory (memory data)
  + lastCategory (memory data)
  + lastStatus (memory data)
  + ProgActivities
  + myTags (memory data)

**processReplicated() function.**

Now this function will be executed via a AsyncBuffer queue:

* An IntervalFunction will run every 30 – 60 seconds and detect changes in the monitored tables (listed in *\_sys\_Trigger\_Tables*) introduced by the respective sqlite triggers.
* For any changes detected (via a timestamp) a cursor is enqueued in a AsyncBuffer with the class and the method to be executed.
* The logic of AsynBuffer executes the class.processReplicated() function pulled from the buffer queue, using the prioritization logic of the buffer.

The code is shown below

IntervalFunctions func, from threading.py

def checkTriggerTables():  
 *""" Checks whether TimeStamp has changed for the rows in \_sys\_Trigger\_Tables. For any changes, enqueues a cursor in  
 replicateBuffer with all the information needed to perform checks and updates of the memory data structures  
 associated to each of the tables.  
 """* try:  
 con = connCreate(MAIN\_DB\_NAME)  
 except (sqlite3.Error, sqlite3.DatabaseError):  
 return None  
 else:  
 if not isinstance(con, sqlite3.Connection):  
 return None  
 cur = con.execute("SELECT DB\_Table\_Name, TimeStamp FROM \_sys\_Trigger\_Tables; ")  
 with con:  
 if not isinstance(cur, sqlite3.Cursor) or cur.rowcount <= 0:  
 return None  
 cur\_data = cur.fetchall()  
 con.close() # NEED to close() here. Exiting with con: will call commit(), but NOT con.close()  
 if cur\_data:  
 tbl\_tstamps = {cur\_data[i][0]: cur\_data[i][1] for i in range(len(cur\_data))} # {DB\_Table\_Name: TimeStamp, }  
 else:  
 tbl\_tstamps = {}  
 if not tbl\_tstamps:  
 return None  
  
 for k in tbl\_tstamps: # k is DB\_Table\_Name  
 if tbl\_tstamps[k] != checkTriggerTables.tstamps\_last\_values[k]:  
 print(f' hhhhhhhhheeeeeeeeeey Trigger Tables: AQUI ESTOY con {k}: {tbl\_tstamps[k]}!!!--------------------')  
 if k in tables\_and\_methods:  
 the\_method = tables\_and\_methods[k]  
 checkTriggerTables.tstamps\_last\_values[k] = tbl\_tstamps[k] # updates TimeStamp in last\_values dict.  
 if callable(the\_method):  
 the\_object = tables\_and\_binding\_objects.get(k, None)  
 if the\_object:  
 replicateBuffer.enqueue(the\_object=the\_object, the\_callable=the\_method)  
 else:  
 replicateBuffer.enqueue(the\_callable=the\_method)  
 else:  
 return None  
 return True  
  
checkTriggerTables.tstamps\_last\_values = sdict.fromkeys(tables\_and\_binding\_objects) # {DB\_Table\_Name: TimeStamp,}

processReplicated() func in krnl\_abstract\_class\_animal.py

**16Aug23. System Updates. Effects of replication on system updates.**

As of now, 2 system updates are identified:

* ~~Category updates driven by time elapsed.~~ Removed: Categories will be computed in real time, on demand. See below.
* ProgActivities created by time elapsed.

Given Marmot replicates all tables and all fields, system updates generated in each node (which are identical in nature) will be replicated to the rest of the nodes. In this way, 1 Category update in a system with 3 nodes will generate 3 sets of records. The same occurs with the creation of 1 ProgActivity in each node.

**20Aug23. Updating individual objects in memory.**

In addition to this problem of multiple replications, there’s the problem of the need to access individual records in DB when one has to update individual objects in memory (specific Animal or Tag objects, for instance). This creates the need to access the Object’s Link table, which will be usually very large, with the time consumption that carries along.

This drives the initial decision to avoid accessing individual objects in memory at all cost.

To solve this problem:

* **Categories:** Categories will only be set upon entry (birth, purchase, etc) and on Castration.

All categories triggered by time will be computed on demand (real time, same as age). This eliminates the problem of multiple replications of system-generated Category updates in every node altogether.

TODO: Modify getCategory(), getInventory(), getStatus(), getLocalization() to obtain its values only from DB, ignoring the memory values.

* **ProgActivities:** Thinking about it.As of now:
  + Allow for ALL repeated records (1 set of records per node) for each ProgActivity to be replicated across the system for ProgActivity generated by the System (Sanidad, etc).

This approach would be implemented with exactly the same replication management that is used for Animales, Caravanas, Personas, etc., meaning:

1. 1 Trigger is fired upon INSERT/UPDATE of closure of a ProgActivity and updates the row in table in table \_sys\_Trigger\_Tables, corresponding to the ProgActivities RAP (and an additional row corresponding to ‘Programacion De Actividades’ if necessary).
2. 1 IntervalTimer function that checks for changes in the row(s) of \_sys\_Trigger\_Tables associated to ProgActivities and executes the update logic.

This logic will:

1. NOT work with any memory data structures. ProgActivities will not hold any memory data structures.
2. The ProgActivity will be closed independently using the existing *\_paMatchAndClose()* and *\_\_checkFinalClose*() methods, said closure done by the User of by the System. When a closure is detected, the code will search and close ONLY the ProgActivities created by that node (based on DB\_ID field value).

Only the node recording (the node “executing”) the closing activity will perform the close of the ProgActivities. The rest of the nodes will only replicate the resulting records with the closure recorded in them.

The logic also will follow this principle: *For every executed Activity (created by the node or received in the node via Replication), it will close ONLY the ProgActivities for each object created by the node running the code, which comply with the closing criteria.*

~~This way:~~

* + - ~~Closes all the duplicated ProgActivities records generated by the System in every node, and closes them with the right Closing Status (this is VERY important, since any ProgActivity left open for a time long enough will be closed as “Expired – Not performed” by the System). So ProgActivities actually performed must be closed as “performed”.~~
    - ~~This closure of a ProgActivity with the right closing status is done at any time, whenever the ProgActivity records reach the closing node via replication. This is described in c) and d).~~

1. The logic will handle the possibility that at the time of closure not all nodes may be replicated, this resulting in a replicated record coming in AFTER the ProgActivity has been closed (but only for all duplicated records existing in the node database at the time of closure). Then, the code in the IntervalFunction will continue to check periodically to close these “late” replicated records that may appear after the ProgActivity is closed.

Using the closure datetime of the executed Activity and the closing windows of the ProgActivities to be closed, the code must be able to assign the right closing status, independently of when (or how late in time) the replication of a ProgActivity record is done on the ‘closing’ node (closing node is the node executing an Activity which, as defined above, is the only node enabled to close ProgActivities that match the executed Activity).

1. The code will then traverse all records added in *RAP* and *ProgActivities Status* tables in each node, check for records added from other nodes due to replication update them with closure data when those records conform to the closing criteria of records already closed by an Activity executed by the node.
   * It will use a “sliding window” for executed Activities in order to limit the number of records to analyze in the RAP table: the concept is to use the “expiration date” for a ProgActivity to be closed as the limit date to search for executed Activities since any ProgActivity that goes past its “expiration date” without closing is closed automatically by the system. Then, for any executed Activity, method \_paMatchAndClose() will search for ProgActivities in that window and will close all those that match the criteria (regardless of where they come from, effectively closing any duplicated, system-generated ProgActivities that may have been generated in a different node).
   * Records in RAP closed by \_paCheckFinalClose() will be closed only by the owner node: each node closes only its own records in RAP.
2. Additionally, code from another function will run regularly closing any ProgActivities left open after expiry date (By design, once a ProgActivity goes past expiry date, NO new Activity will match it for closure).

**18Aug23. Terminal ID Table:**

A table named \_sys\_terminal\_id is added which lists only 1 row containing the UUID for the database.





This UUID is incorporated via a TRIGGER AFTER INSERT in tables Animales, Caravanas, Personas, Geo Entidades, Dispositivos, Registro De Actividades, Registro De Actividades Programadas.

This field aids in the logic of *processReplicated(), processTags() and processProgActivities()* methods.

**IMPORTANT:** When this table is replicated, all other nodes’ DB\_ID will be appended to the table in subsequent rows. For each node in the system, a row with the respective DB\_ID will be inserted in this table.

*The only way to identify the local databases ID is by accessing ROWID=1, which is done using the LIMIT 1 clause* in the sql statement used to pull db ID.

**21Aug23. Background tasks on ProgActivities.**

1. An IntervalFunction running every 6 – 8 hours will execute function *processProgActivities()*.
2. processProgActivities in turn call class methods in active ProgActivities classes (ProgActivityAnimal, ProgActivityPerson, etc) and for each of them will run all “cleanup” and administrative methods that are required for progActivities.

These classmethods will be invoked via a cursor buffer, same as for Triggers. (Could use the already existing *ProgActivityCursor* buffer.).

One of the administrative functions for ProgActivities will be *closeStaleProgActivities().*

1. *closeStaleProgActivities* will use the sliding window mentioned above (items 2a – 2d) to implement in every node the closure of ProgActivities that have been replicated in the node ***AFTER*** an executed Activity closed them.
2. Additional cleanup functions may be needed for ProgActivities, to be run periodically in the processProgActivities() function. To be defined.

**21Aug23 / 24Aug23. Update of Tags.**

Tag replication will be handled (same as Animal, Person, etc) by a dedicated processReplicated() classmethod in Tag class.

Tag structures (myTags, \_\_identifiers, etc) will be updated in Animal.processReplicated method, using \_\_identifiers. This is the most efficient way to manage assigned tags updates.

The logic for tags will be implemented via Tag.isAssigned(self) method***.*** This method will return the object that the tag is assigned to, or None of the tag is not assigned to an object.

The code implements the logic below:

1. During initialization, all tag initialization routines assign a value to Tag attribute *\_\_assignedTo.* This attribute will hold 1 and only 1 object (of any nature) owns the tag.
2. Tag object calls isAssigned()
   1. If isAssigned value is != None, returns that value (an object of any nature).
   2. If isAssigned is not found, the initialization routine is run again for the class (Animal, Device, etc.) and Tags are reloaded from [Caravanas] table using the existing upload functions. This function will update \_\_registerDict with all tags added and removed up to the point of running the query.
   3. isAssigned() is called again and returns whatever value the attribute \_\_*assignedTo* holds after the call in b. (an object of any nature, or None)
3. Sdf

**Uniqueness of Tags.**

Tags are required to be unique only within an object class. With this, there can be repeat tagNumber values for different object classes. Ex.: tag numbers must be unique ***within*** Bovine, Caprine and Devices classes, but a certain tagNumber can be Active both for Caprine and Bovine, as the object can be identified univocally in this case.

With this, the test for tag assignment (call to isAssigned method) must be implemented as a classmethod for the different classes using Tags.

**25Aug23. On the asynchronous nature of database replication.**

Marmot (and other replicators) work on “eventually replicated” databases model, stating that all records will ***eventually, at one point in time***, be replicated***.***

This means that the replication time and the replication order is not controlled and cannot be predicted.

With this operating condition, the system adopts **a “work with what’s available” principle**, meaning that at all times the system will pull and work with the records available in the database. As more records are made available to a particular node via replication, the system code in that node will start using them.

This also means that, by design, any efforts to pull “missing” or “not replicated” records to perform any operations should be avoided at all costs, as it adds a complexity in the logic and instability in the system that work against system integrity.

In this scheme, when data or records are “missing” due to replication lag, the only action to execute is run the memory-update routines for the object to account for the possibility of data having been replicated in database, but not yet updated in memory by the memory-updating routines (processReplicated() in most cases).

**26Aug23. Resuming ProgActivities.**

User-generated ProgActivities closure:

* ~~The node executing and Activity will close ALL ProgActivities that match conditions, that are available in the node at Activity execution time.~~ -> See below.
* All nodes will run a Activity.processReplicated() method for executed Activities. The method will matchAndClose() ONLY ProgActivities created by the local node that are a match for the replicated executed Activity.

TODO: For this, a “*this\_node\_only = False”* flag must be included in method \_paMatchAndClose() to filter only ProgActivities generated by the node, using the DB\_ID field in the ProgActivity record.

This way, the executing node can close all ProgActivities that are available that are then later replicated without duplications or ambiguity.

And for replicated executed Activities, the closing of only the “local” ProgActivities (ProgActivities generated by that same node) limits the closure possibilities for a given ProgActivity, independent of its replication time, and prevent multiple closure by different nodes of the same ProgActivity when replication is not immediate.

**Now on to system-generated ProgActivities:**

These progActivities are generated for the same objects, by all nodes. It is assumed that they are generated more or less at the same time (similar or equal creation timestamp).

With this, the situation is that ***for each ProgActivity n instances are created when n nodes are running in the system.***

The challenge is to use the same logic used in user-generated ProgActivities to matchAndClose these ProgActivities 🡺 The solution seems to be:

* The node executing and Activity will close ONLY ProgActivities created by the same node that match conditions. This assures that the ProgActivity to close will always be there (it was created by the node).
* All nodes will run an Activity.processReplicated() method for executed Activities. The method will matchAndClose() ONLY ProgActivities created by the local node and that are also a match for the replicated executed Activity.

TODO: Implement a “*this\_node\_only = True”* flag in method \_paMatchAndClose() to filter only ProgActivities generated by the node, using the DB\_ID field in the ProgActivity record.

Defined criteria:

*ProgActivities: each node closes only its own ProgActivities (as defined by field Terminal\_ID) and NO other ProgActivities. The closure is done based on executed Activities that match the ProgActivity conditions, irrespective of the origin of the executed Activity (same node or acquired via replication).*

*Executed Activities: Also, each node processes ALL executed Activities (coming in table RA) with the rationale that a closing Activity for one of the node’s ProgActivities can be generated in* ***any node*** *in the system.*

This way, all ProgActivities are treated equal, regardless of whether they are user o system-generated.

The replication process works as follows:

1. A ProgActivity is created by a node (user o system-generated).
   1. The ProgActivity is eventually replicated to other nodes, at different times.
2. A node (any node) executes an Activity that is a match to close the ProgActivity in 1.
   1. The node (any node) runs *\_paMatchAndClose()* that checks for matches in its list of ProgActivities with and closes any ProgActivity created by the node that is a match.
3. The executed activity will be replicated to the rest of the nodes eventually and, for each replication:
   1. Each node will run an *Activity.processReplicated()* method that looks for replicated executed Activities and runs them through the \_paMatchAndClose() method, closing any ProgActivities that are a match and that were created by that node.

To achieve this, the logic in processReplicated looks only for INSERTed records. Those INSERT records are executed Activities generated on other nodes. Then, *\_paMatchAndClose()* is run for each of those INSERTed records.

1. The closed ProgActivities will be replicated through the system and eventually reach all nodes, with their proper and correct closing data (including closing Status, etc).
2. ProgActivity closure due to running past expiration date will also be limited to the ProgActivities created by the node to keep consistency with the logic described here.

The replication process then will take longer due to the executed Activity and ProgActivity replication back-and-forths, but will be significantly simpler and much more robust in terms of system integrity.

**30Aug23. Updates to Triggers. *Adding Execute Data, Excluded Fields* fields to Link Tables.**

INSERT triggers are updates with a clause to fire the UPDATEs of \_*sys\_Trigger\_Tables.TimeStamp* field ONLY when records from external nodes are INSERTed.

Trigger works as follows:

UPDATE:

return f'CREATE TRIGGER IF NOT EXISTS "Trigger\_{dbTblName}\_UPDATE" AFTER UPDATE {col\_list\_str}' \  
 f' ON "{dbTblName}" FOR EACH ROW BEGIN ' + \  
 (f' UPDATE "{dbTblName}" SET "Record UPDATE" = IFNULL(old."Record UPDATE",'   
 f' {randrange(1000, 5000) \* randrange(100, 1000)}) + 1 WHERE ROWID=new.ROWID;'  
 if "Record UPDATE" in fldNames.values() else '') + \  
 (f' UPDATE \_sys\_Trigger\_Tables SET TimeStamp=DATETIME("now","localtime"), Last\_Updated\_By="{MAIN\_DB\_ID}"'  
 f' WHERE DB\_Table\_Name="{dbTblName}";'  
 if dbTblName in tbl\_list else '') + \  
 f' END; '

**INSERT:**

return f'CREATE TRIGGER IF NOT EXISTS "Trigger\_{dbTblName}\_{operation}" AFTER {operation} ON "{dbTblName}"' \  
 f' FOR EACH ROW BEGIN' + \  
 (f' UPDATE "{dbTblName}" SET DB\_ID=(SELECT DB\_ID FROM \_sys\_db\_id LIMIT 1) WHERE ROWID=new.ROWID; '  
 if operation == 'INSERT' else '') + \  
 (f' UPDATE \_sys\_Trigger\_Tables SET TimeStamp=DATETIME("now","localtime"), Last\_Updated\_By="{MAIN\_DB\_ID}"'  
 f' WHERE DB\_Table\_Name="{dbTblName}" AND new.DB\_ID != "{MAIN\_DB\_ID}";'  
 if dbTblName in tbl\_list else '') + \  
 f' END; '

IGNORE on UPDATE of \_sys\_db\_id record #1:

trigger\_sql = f'CREATE TRIGGER IF NOT EXISTS IGNORE\_UPDATES BEFORE UPDATE ON \_sys\_db\_id FOR EACH ROW BEGIN ' \  
 f'SELECT RAISE(IGNORE) WHERE old.ROWID=={rowid}; END; '

**This goes to ProgActivities.**

**30Aug23. Proper Field Names. Compare\_Index, getFldCompare().**

**Proper Field Name: a string of the form *“tblName.fldName”* where both tblName and fldName are key names (all tables start with ‘tbl’, all fields start wit ‘fld’).**

A Column Compare\_Index is added to the \_sys\_Fields table. The column defines indices for comparable fields: 2 fields (from different tables) with equal value in Compare\_Index column can be compared. This comparison is intended to be performed by ProgActivities methods that must usually match values from fields in ProgActivities tables with values in regular, executed Activities tables, where the field names are not necessarily equal.

Additionally, fields in ProgActivities dictionaries will be stored as *“tblName.fldName”* string, when required. This nomenclature allows to access the correct field in table \_sys\_Fields, and from it, access the Compare\_Index for the fields in order to perform (or not perform) comparison of field values.

The function code is below:

def getFldCompare(fld1: str, fld2: str):  
 *""" Returns True if fld1 is comparable to fld2 (they share the same Compare\_Index value).* ***@param*** *fld1: tblName1.fldName1 (str)* ***@param*** *fld2: tblName2.fldName2 (str)* ***@return*** *True if fld1 and fld2 share Compare\_Index values. Otherwise False.  
 """* split1 = fld1.split(".")  
 split2 = fld2.split(".")  
 tbl1, fld1 = (split1[0], split1[1]) if len(split1) == 2 else (None, None)  
 tbl2, fld2 = (split2[0], split2[1]) if len(split2) == 2 else (None, None)  
 if any(not i for i in (tbl1, fld1, tbl2, fld2)):  
 return False  
  
 fldName1 = getFldName(tbl1, fld1, 1)  
 fldName2 = getFldName(tbl2, fld2, 1)  
 fld1CompIndex = fldName1[2] if isinstance(fldName1, (list, tuple)) else None  
 fld2CompIndex = fldName2[2] if isinstance(fldName2, (list, tuple)) else None  
 if isinstance(fld1CompIndex, int) and isinstance(fld2CompIndex, int):  
 return fld1CompIndex == fld2CompIndex  
 return False

Compare\_Index is used in \_\_isClosingActivity() logic as the 1st option to check when values to compare ***are* proper fields** (that is, string of the form “*tblName.fldName*”)

If getFldCompare() returns false, then checking for comp() method or using “shortname” properties are executed.

The comparing excerpt of the code is below - krnl\_abstract\_class\_activity.\_\_isClosingActivity ():

# *TODO(cmt) COMPARISON RULES. keys in execute\_fields compare as follows:* # *All fields in d1Comp are compared with the data in d2Comp: If one match with d2 gives False, the* # *comparison is False. If any of keys in d1Comp is not present in d2 -> also False.* # *1) If 'fld' particle in key it's DB field name, uses compare() function. Else, flattens d2 to a list and:* # *2) if a dict is found in d2Flat, attempts comp method: outer\_obj.getattr(outer\_obj,d1Comp[k]).comp(d[k]).* # *If that fails, res is = False (comp() is not implemented for k and k is NOT a fldName, hence False).* # *2.a. First, uses compare() with fields of the form "tblName.fldName", using getFldCompare().* # *2.b Else if no dict was found in d2Flat then k may be a db field name: searches for keyname using the* # *"shortname" property to fetch a field name particle, pulls the compare value (comp\_val) and runs* # *outer\_obj.getattr(outer\_obj, d1Comp[k]).comp(comp\_val).* d2Flat = list(nested\_dict\_iterator\_gen(d2))  
 matchResults = {}  
 for k in d1Comp:  
 # Here k can be in the form of "fldName" or "tblName.fldName".  
 # if form "tblName.fldName" is detected, must prioritize comparison using getFldCompare().  
 # Priority of execution:  
 # 1) if dkey from dic == k OR dkey is contained in k, dic2 is populated with that dic from d2Flat.  
 # 2) if 1 fails, executes getFldCompare(k, dkey). This results in dic2 populated or dic2={} if fails.  
 dic2, dic2val = next(((dic, dic[dkey])for dic in d2Flat for dkey in dic if dkey.lower().strip()  
 in k.lower()), ({}, VOID))  
 if 'fld' in k.lower(): # si k contiene fld -> NO es atributo y NO tiene comp() definido: usa compare()  
 if not dic2 and "." in k:  
 # Checks comparable fldNames. Here, k and dkey must be both VALID "tblName.fldName" strings.  
 dic2val = next((dic[dkey] for dic in d2Flat for dkey in dic if getFldCompare(k, dkey)), VOID)  
 res = compare(d1Comp[k], dic2val) # if not d but 'fld' in k.lower() => then res = False.  
 elif dic2:  
 try:  
 res = getattr(outer\_obj, k).comp(dic2val) # executes successfully if k implements comp()  
 except (AttributeError, NameError, TypeError, KeyError, ValueError):  
 res = False # if not: it's not a field, it's not an attribute => result = False.  
 else:  
 # NO key in d2Flat could be matched to k: pulls keyname using shortName property in Activity class.  
 comp\_val = next((dic[dkey] for dic in d2Flat for dkey in dic if  
 getattr(outer\_obj, k).shortName.lower() in dkey.lower().strip()), None)  
 try: # executes if conditions[k] implements comp().  
 res = getattr(outer\_obj, k).comp(comp\_val)  
 except (AttributeError, NameError, TypeError, KeyError, ValueError):  
 res = False  
 matchResults[k] = res  
  
 print(f' \*\*\*\*\* Compare Results = {matchResults}; execution date: {matchDates}', dismiss\_print=DISMISS\_PRINT)  
 return all(j is True for j in matchResults.values()) and matchDates  
return False

**17Aug23. DataStreams**.

Generally speaking, data records coming from real-time acquisition devices (Geolocation, weather data, temperature data, general sensors, etc.).

The data generated may or may not be associated to the main database structure. It may or may not be attached to objects in the system, depending on the specific use of the data.

1. *Look into managing DataStream will be store on a single DataStream Database (named* ***GanadoSQLite\_DS****) in every node, with dedicated table structures to allocate and manage all the Data Streams.*
2. *The* ***GanadoSQLite\_DS*** *database will also be replicated by Marmot. The logic will work as follows:*
   1. *Every device broadcasting DataStreams will transmit every datastream record* ***only once*** *to whichever node is found first and mark those records as transmitted, so that they are not broadcasted again.*
   2. *The node receiving the datastream record(s) will replicate those records with Marmot.*
   3. *In order to optimize data availability in certain environments with long ranges, a dedicated node could be implemented that is in permanent connection with all DataStream Devices. These devices transmit to that ‘always-on’ node (a device linked to a radio transceiver that makes the connection link with the DataStream Devices), and the node replicates the data to the “user” nodes (mobiles, tablets, etc.).*
   4. *If needed the DataStream records can be assigned a UUID as part of their0 processing for ease of identification across all nodes once they are replicated.*
3. *Implement a registry for databases names assigned to each device that stores basic parameters for each database. (Database names, subscriptions, etc.). This registry should be shared among server and all registered devices.*

The structure for DataStreams are a “always-on” device that receives the datastream via radio link (4/5G, Satellite).

That device stores the raw data of all streams (ALL DATA RECEIVED) in a local storage, with proper backup.

The DataStream Manager Application processes that raw data according to defined criteria and writes that data to a 2nd database that is replicable (GanadoSQLite\_DS.db)

This replicable database is then replicated by Marmot to all devices for their use.

**Subscriptions**: Local databases in the devices may subscribe to receive DataStream data from Stream databases. The subscriptions will be listed in a dedicated database definition table and will include the DataStream database names the main db is subscribed to.

Subscriptions should also work for System-generated notifications such as Sanidad, Category, System Status, Errors, etc. -> See how this can be implemented.

* Need to look into how subscriptions will work and how and what type of DataStream records will be shared with the main db.

**01Dec23: Database Record Duplication with new paradigm of ‘No memory data structures’.**

**Record duplication management will work as follows:**

* **Duplication management ONLY implemented for Animales, Caravanas, Dispositivos, Personas, Proyectos tables.**
* **All duplicate records are left on the DB for integrity. No records should be deleted.** *This is critical to ensure proper change-cascading across tables when records are modified.*
* SQLite Triggers should update \_sys\_Processed\_Activities ONLY when a duplicate field is detected, to minimize execution cycles of the processReplicated() background function.
* The Trigger should update the DB ID field adding the current DB UID to as the last item to the DB ID list (JSON field).
* **Identifiers**: In order to facilitate the use of triggers to initiate replication processing, the use of the Identifiers (“Identificadores”) field is critical*: the contents of Identifiers field will be used to determine whether duplication occurs.*

**Specific Identifiers:**

* + **Animales:** JSON List. A list of all Tag UIDs assigned to the Animal Object.
  + **Caravanas:** Generated Field. TEXT. Concatenation “Numero Caravana” || “ID\_Tecnologia De Caravana” fields.
  + **Geo:** Generated. TEXT. A Concatenation of the Entity Name and 1 or 2 containers immediately enclosing (the next 2 higher Localization Levels). Ex.: City Name&Department Name&Province Name, Province Name&Country Name, Establecimiento&Department Name& Province Name.

*A very straight forward way of setting this is making it equal to the “Containers” field.*

* + **Personas:** Generated Field. Concatenation of valid IDs defined for the person.

*Must be generated by the python Person constructor.*

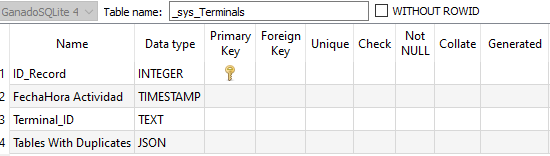
* + **Dispositivos**

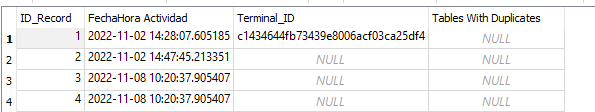
With this, the verification of values in the trigger code to check if some value contained in the Identifier field becomes much more manageable.

The advantage of using Generated Columns is that if any of the components of the column changes, Identifiers is updated automatically, future checks will yield the right result in terms of detecting duplication and since the rest of the row remains unchanged, the record remains unchanged for use by other parts of the program.

* The python logic (via function processReplicated()) pulls the data from the \_sys\_Terminals table and runs the logic to update DB records ONLY associated to the local device (using its value of Terminal\_ID to update only its associated record in \_sys\_Terminals table).

Once processReplicated() logic is executed, the DB\_Table\_Name for which the update was made is removed from the list in the DB ID Field of \_sys\_Table.

****



Tables With Duplicates is a JSON List field.

In this table, TimeStamp may no longer be needed.

Every device updating the table via Triggers will append the Table Name of the table being modified to the list, and items will be processed sequentially (FIFO) by the different devices.

**See how this can be done by the Trigger itself, ONLY when a duplicate value is detected**: This will be done by dedicated triggers that fire only when specific conditions for duplication are met (these conditions will vary from table to table).

Then, the logic in processReplicated() will first turn the list into a set(), process the data, remove the DB ID element of the set and re-write the updated set onto “Tables With Duplicates” field.

**2-dec-23. Managing duplicates**

***Terminal\_ID field*** *replaces* ***DB\_ID*** *in all tables, all code, for all purposes*.

**Original Record**: A record for an item object (Animal, Device, Tag, Person, Device) ***first created*** (in time, that it, with the earliest TIMESTAMP date).

**Duplicate Record:** A record for an item object (Animal, Device, Tag, Person, Device) sharing 1 or more identifiers (within a set of identifiers, as defined for the object) that was created ***after*** the Original Item (a later TIMESTAMP date).

*Duplicate records refer to the same physical object and are created by independent terminals when operating without synchronization/db replication.*

**Triggers defined:**

* **Object Tables, INSERT:** Trigger ON INSERT. Initializes the Terminal\_ID field for all records ONLY when first created. Later replications (with Terminal\_ID already set) have no effect on the Terminal\_ID value. Managed by SQLITE.
* **Object Tables, Duplication:** Trigger ON INSERT. Flags Original and Duplicate records, for Duplicates Management. See if resolution of Duplicates can be resolved without resourcing to python, just by the TIMESTAMP value of the record. Flagged by SQLITE in table \_sys\_Terminals. Managed by SQLITE??
* **Object Tables, UPDATE:** Trigger ON UPDATE OF “UID\_Objeto”. Updates records on all tables linked to the UID\_Objeto value. Managed by SQLITE.
* **See if ProgActivities, Activities require additional triggers.**

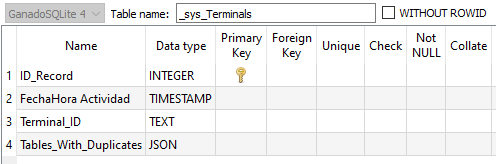
Animal, Tag, Geo, Person, Device, Project are subject to creating duplicate instances for the same physical object when different terminal devices are not linked (via wifi, Bluetooth, 5G ,etc).

In this case, when the devices eventually link, these devices will end up with to separate records in the respective table (Animales, Caravanas, etc) that refer to the same physical item, ***but these records will have a different UID***.

The system must be able to detect these duplicates based on the check of a unique identity for the object and resolve the issue for all terminals in the system. (a terminal is a computing device that runs the application: mobile, pc, laptop, etc.).

Identifiers for different types of objects are listed above.

* For integrity of the system and considering the limitations imposed by how Marmot does the replication, a table \_sys\_Terminals needs to be defined with the following structure



The purpose is that each terminal writes/updates ONLY the row assigned to it, and no other rows.

* Each terminal will ONLY update its local records and ONLY when there’s a flagging for update in its assigned row in the \_sys\_Terminals table.

In this way, all terminals update their records and manage record duplication logic independently and asynchronously and now updates are overwritten of missing if a certain device spends long periods of time offline.

* Duplication Logic (function processDuplication()): The logic must identify the original records and 1 or more duplicate records.

For this:

1. Triggers will update the field \_sys\_Terminals.Tables with the table name where duplicates are detected.
2. When launched, the processDuplication() function will query the row belonging to the terminal running the function and if data is found will process the duplication for the table (or tables) listed in the \_sys\_Terminals.Tables (which is a JSON list in SQLite).
3. A general logic is:

Duplicate records occur when more than 1 record in a Object Table (Animal, Caravana, Person, Geo, etc) share 1 or more identifiers. On creation they also have a different value of UID\_Objeto. They are generated by separate Terminal operating without a synchronizing link (wifi, bluethooth, 5G, etc).

To solve this situation, duplicates are detected and the duplicates have their UID\_Objeto column updated with the value of the original record. This is to enable the operation of the system with the use of a unique value of UID\_Objeto for each object item defined.

*As a result, UID\_Objeto is no longer a UNIQUE field. All other tables with values linked to this field will have to be updated via an UPDATE TRIGGER.*

To find duplicates:

* 1. Find the record with the earliest (oldest) TIMESTAMP in the table with duplicates. That’s the original.
  2. Copy the original UID\_Objeto value from the original to all the duplicates records found.
  3. Separately, the function that retrieves a record by UID in tables with duplicates (*getObject* functions) must search for the UID passed and *return the record with the earliest (oldest) TIMESTAMP.* ***This is important to be consistent with all the record data that is used to create the memory object every time*.** This is transparent to the user.

1. 2 ways of generating a duplicate for Animales:
   1. INSERTING a record with a repeated UID\_Object (trivial). UID\_Objeto are UNIQUE fields, so this will be rejected automatically. No need for Triggers.
   2. INSERTING a record with at least 1 of its identifiers present as an identifier in some other record in the table. This one will be trickier to resolve. -> May be checking json sub-strings. See json interface functions in sqlite.

This will lead to 2 different triggers being created to detect these situations.

The trigger below is for case b. -> Repeat identifiers.

***03Dec23:***

***Duplication of records: Triggers DETECT duplication and report it.***

***System background functions process the duplication as they are reported by the Duplication Trigers.***

***Example of Duplication mgmt. Trigger (Animales):***

*CREATE TRIGGER IF NOT EXISTS Trigger\_Animales\_Check\_Duplicates AFTER INSERT ON Animales FOR EACH ROW*

*WHEN*

*(SELECT DISTINCT UID\_Objeto FROM Animales, json\_each(Animales.Identificadores)*

*WHERE Animales.ROWID != NEW.ROWID AND json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND*

*("Salida YN" == 0 OR "Salida YN" IS NULL))*

*BEGIN*

*UPDATE \_sys\_Terminals SET "Tables With Duplicates" = json\_insert("Tables With Duplicates", '$[#]', "Animales"||'-'||NEW.ROWID)*

*WHERE Terminal\_ID = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1);*

*END;*

*This Trigger accomplishes a lot with a few lines:*

1. Searches for any occurrence of ANY of the identifiers passed during the INSERT (NEW.IDentificadores\_str) in the FULL list of identifiers present in [Animales.Identificadores]. Each value in this col is a list of identifiers, in JSON form.
2. If none of the identifiers just passed are not present (the WHEN clause fails), exits without doing a thing.
3. If any of the identifiers is present in any of the rows, this means that the identifier is repeated, so the logic to determine duplicate Animal must be triggered. This is done by setting the field *\_sys\_Terminals.“Tables With Duplicates”.*
4. Only the row that belongs to the current Terminal (based on Terminal\_ID value) is modified.
5. The field *\_sys\_Terminals.“Tables With Duplicates*, which is a JSON list, is appended with a string of the form “TableName-ROWID”. In this way, when the logic pulls this field to process duplicate Animals for the table TableName, the values of ROWIDs to go and pull the data are already provided.

Just in case, the python logic must contemplate the case when ROWID is not present in the value red, or is not valid, in which case it must re-do the search that SQLite does with the json\_each() function and json\_each.value attribute.

All in all, pretty nice.

A much simpler for Geo solution is to use a trigger that uses COUNT() and count the number of times a full Identficadores string appears in Identificadores column. Same for Caravanas. As shown below.

Trigger for Caravanas Duplication:

*CREATE TRIGGER IF NOT EXISTS Trigger\_Caravanas\_Check\_Duplicates AFTER INSERT ON Caravanas FOR EACH ROW*

*WHEN*

*(select (SELECT COUNT(\*) FROM Caravanas WHERE Identificadores == NEW.Identificadores AND ("Salida YN" == 0 OR "Salida YN" IS NULL)) > 1)*

*BEGIN*

*UPDATE \_sys\_Terminals SET "Tables With Duplicates" = json\_insert("Tables With Duplicates", '$[#]', "Caravanas"||'-'||NEW.ROWID)*

*WHERE Terminal\_ID = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1);*

*END;*

A completely equivalent ones for Geo Entidades.

For personas, equivalent to Animales and Devices.

The triggers do a lot more: ***They eliminates the need of python code and implement the full logic for managing duplicates within the SQLite code. The same concept to be used for the Duplication mgmt. triggers of all the other tables.***

**Animal, Person, Device**: The trigger fires when 1 (or more) of the defined “Identificadores” is a match. For this the json functions are very handy as they traverse the full “Identificadores” column, matching each individual item in each of the JSON arrays. The trigger using json functions is below:

*CREATE TRIGGER IF NOT EXISTS Trigger\_Animales\_Check\_Duplicates AFTER INSERT ON Animales FOR EACH ROW*

*WHEN*

*(SELECT DISTINCT UID\_Objeto FROM Animales, json\_each(Animales.Identificadores)*

*WHERE Animales.ROWID != NEW.ROWID AND json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND*

*("Salida YN" == 0 OR "Salida YN" IS NULL))*

*BEGIN*

*UPDATE Animales SET UID\_Objeto = (SELECT DISTINCT UID\_Objeto FROM (SELECT DISTINCT UID\_Objeto, "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND ("Salida YN" == 0 OR "Salida YN" IS NULL))*

*WHERE*

*"FechaHora Registro" == (SELECT MIN("FechaHora Registro") FROM (SELECT DISTINCT UID\_Objeto, "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores) WHERE json\_valid(Animales.Identificadores) AND*

*INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND ("Salida YN" == 0 OR "Salida YN" IS NULL))))*

*WHERE Animales.ROWID == NEW.ROWID;*

*UPDATE \_sys\_Terminals SET "Tables With Duplicates" = json\_insert("Tables With Duplicates", '$[#]', "Animales"||'-'||NEW.ROWID)*

*WHERE Terminal\_ID = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1);*

*END;*

**Tag, Geo**: Because of the definition of the Identificadores column, the trigger fires when the **value** in Identificadores is equal to the comparison value. The function of the trigger is the same (UPDATE the value in “UID\_Objeto” column). However, in this case there’s no need for json functions. The code for the final triggers is below:

*CREATE TRIGGER IF NOT EXISTS Trigger\_Caravanas\_Check\_Duplicates AFTER INSERT ON Caravanas FOR EACH ROW WHEN*

*(SELECT (SELECT COUNT(\*) FROM Caravanas WHERE Identificadores\_str == NEW. Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL)) > 1)*

*BEGIN*

*UPDATE Caravanas SET UID\_Objeto = (SELECT DISTINCT UID\_Objeto FROM (SELECT DISTINCT UID\_Objeto, "FechaHora Registro" FROM Caravanas*

*WHERE Identificadores\_str == NEW. Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL))*

*WHERE"FechaHora Registro" == (SELECT MIN("FechaHora Registro") FROM (SELECT DISTINCT UID\_Objeto, "FechaHora Registro" FROM Caravanas*

*WHERE Identificadores\_str == NEW. Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL))))*

*WHERE Caravanas.ROWID == NEW.ROWID;*

*END;*

*CREATE TRIGGER IF NOT EXISTS Trigger\_Geo\_Check\_Duplicates AFTER INSERT ON "Geo Entidades" FOR EACH ROW WHEN*

*(SELECT (SELECT COUNT(\*) FROM "Geo Entidades" WHERE Identificadores\_str == NEW. Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL)) > 1)*

*BEGIN*

*UPDATE "Geo Entidades" SET UID\_Objeto = (SELECT DISTINCT UID\_Objeto FROM (SELECT DISTINCT UID\_Objeto, "FechaHora Registro" FROM "Geo Entidades"*

*WHERE Identificadores\_str == NEW. Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL))*

*WHERE"FechaHora Registro" == (SELECT MIN("FechaHora Registro") FROM (SELECT DISTINCT UID\_Objeto, "FechaHora Registro" FROM "Geo Entidades"*

*WHERE Identificadores\_str == NEW. Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL))))*

*WHERE "Geo Entidades" .ROWID == NEW.ROWID;*

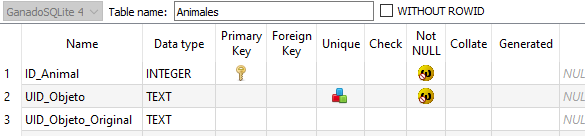
*END;*

**05-Dec-2023**

**Breakthrough concept: UID\_Objeto returns to being UNIQUE.**

**Thow away all of the above comments on modifying** UID\_Objeto **column. It will remain UNIQUE and SEE TO USE A “UID\_Original” Col in tables Animales, Caravanas, Personas, Geo, Dispositivos.**

The idea is to keep the uniqueness of UID\_Objeto across ALL Terminals in the system, at all times as this simplifies the treatment of records and ensures integrity of the database. In particular, no need to fiddle around with triggers for tables that have UID\_Objeto linked as a FK.



As a result, the getObject() methods that pull records based on “Identificadores” or UID\_Objeto will have to return the value in column UID\_Objeto\_Original (as opposed of the previous idea of retrieving the UID\_Objeto with the earliest TIMESTAMP).

*Also,* ***Check\_Duplicates*** triggers need to be modified as follows:

UID\_Objeto\_Original is only modified by triggers and its workings are completely transparent to the API. The API only needs to know when to pull its value.

UID\_Objeto\_Original retains the UNIQUE attribute so it can be linked via FK to all the necessary tables (Link Animales Actividades, Link Animales Actividades Programadas, etc.).

By the way, the lines

*UPDATE \_sys\_Terminals SET "Tables With Duplicates" = json\_insert("Tables With Duplicates", '$[#]', "Animales"||'-'||NEW.ROWID) WHERE Terminal\_ID = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1);*

come back to the Check\_Duplicates triggers in order to resolve inconsistent records, for instance, one Terminal enters the Animal as a MALE and another one as FEMALE, non-matching categories, castration status, localization, dob, etc..

* This will be done by the Python code.
* Will require user input for resolution and the object that eventually is discarded will have “Salida YN” set to 1 to flag that is not a valid record.

Resulting triggers Animales (Personas and Dispositivos) are like this:

*CREATE TRIGGER IF NOT EXISTS* ***Trigger\_Animales\_INSERT*** *AFTER INSERT ON Animales FOR EACH ROW*

*BEGIN*

*UPDATE Animales SET Terminal\_ID = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1) WHERE Animales.ROWID == NEW.ROWID AND NEW.Terminal\_ID IS NULL;*

*UPDATE Animales SET UID\_Objeto\_Original = (SELECT DISTINCT UID\_Objeto FROM (SELECT DISTINCT UID\_Objeto, "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND ("Salida YN" == 0 OR "Salida YN" IS NULL))*

*WHERE*

*"FechaHora Registro" == (SELECT MIN("FechaHora Registro") FROM (SELECT DISTINCT UID\_Objeto, "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND*

*INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND ("Salida YN" == 0 OR "Salida YN" IS NULL))))*

*WHERE Animales.ROWID == NEW.ROWID;*

*END;*

*CREATE TRIGGER IF NOT EXISTS* ***Trigger\_Animales\_Check\_Duplicates*** *AFTER INSERT ON Animales FOR EACH ROW*

*WHEN*

*(SELECT DISTINCT UID\_Objeto FROM Animales, json\_each(Animales.Identificadores)*

*WHERE Animales.ROWID != NEW.ROWID AND json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND*

*("Salida YN" == 0 OR "Salida YN" IS NULL))*

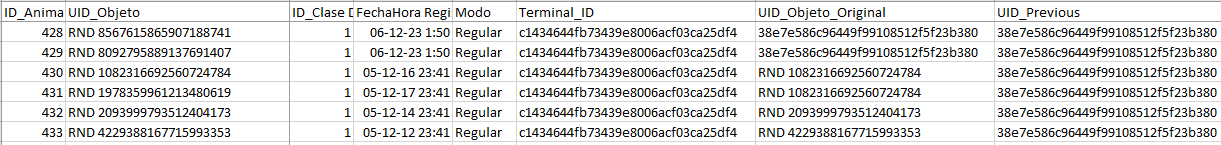
*BEGIN*

*UPDATE \_sys\_Terminals SET "Tables With Duplicates" = json\_insert("Tables With Duplicates", '$[#]', "Animales"||'-'||NEW.ROWID)*

*WHERE Terminal\_ID = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1);*

*END;*

Example of duplicate Animals chain maintained by UID\_Objeto\_Original and UID\_Previous.



UID Previous does not change above because the python Duplication logic is not running.

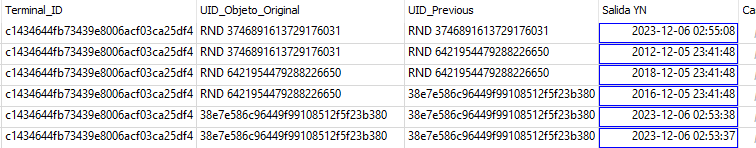
The Python logic must be such that this case is covered: basically, once a difference is found between UID\_Objeto\_Original and UID\_Previous, the whole chain is scanned up and down and the values of UID\_Original are updated to the latest one (the one with the earliest TIMESTAMP):

UID\_Objeto\_Original: ALL values in the chain are set to the one with the earliest TIMESTAMP.

UID\_Previous: For now (and this is to be looked into): Leave UID\_Previous untouched. It doesn’t seem necessary to modify for the logic to work, and leaving them alone keeps the history of Previous UID available.

**This is the chain after fixing the INSERT Trigger:**

TIMESTAMPDate column

****

06-Dec-2023:

**The MAIN rule for replication is: DO NOT MODIFY (UPDATE) records generated/created by other Terminals WHEN IN A LINKED CHAIN (like when in Duplication, as shown above).**

This is because if we do, the modified record will be again replicated to ALL active terminals, which in turn will update the record based in the status of its local chain, which will fire another round of replications of the record, to be again modified by each Terminal’s local chain, and entering an infinite loop of replication.

With this,

**The FINAL logic for Duplication Management is:**

1. **Record chains ARE LOCAL** to the Terminal. Their integrity must be kept at all times.
2. **Triggers should ONLY modify chain records for the local (Mother) Terminal.** The rest of the records added to the terminal database from another terminals (via replication) must be left untouched, keeping the integrity of their local chains.
3. As a result of 2) multiple chains (1 for each active Terminal) with the same original record will co-exist in all terminals after replication.
4. The problem then lies in finding **the original record** for the object (the one with the **earliest TIMESTAMP**) traversing the multiple chains stored in the table. Can be stated as follows:

The Original Record in a set of chains (1 chain per Terminal) is the record with the earliest TIMESTAMP of all records, in all linked chains.

A INSERT Trigger with creation of a chain of UID\_Objeto\_Original, UID\_Previous is follows (Obsolete. Kept here for reference only):

*CREATE TRIGGER IF NOT EXISTS Trigger\_Animales\_INSERT AFTER INSERT ON Animales FOR EACH ROW*

*WHEN NEW.Terminal\_ID IS NULL OR NEW.Terminal\_ID == (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1)*

*BEGIN*

*UPDATE Animales SET Terminal\_ID = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1) WHERE Animales.ROWID == NEW.ROWID AND NEW.Terminal\_ID IS NULL;*

*UPDATE Animales SET UID\_Objeto\_Original = (SELECT DISTINCT UID\_Objeto FROM (SELECT DISTINCT UID\_Objeto, "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND ("Salida YN" == 0 OR "Salida YN" IS NULL))*

*WHERE*

*"FechaHora Registro" == (SELECT MIN("FechaHora Registro") FROM (SELECT DISTINCT UID\_Objeto, "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND*

*INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND ("Salida YN" == 0 OR "Salida YN" IS NULL))))*

*WHERE Animales.ROWID == NEW.ROWID;*

*UPDATE Animales SET UID\_Previous = (SELECT DISTINCT UID\_Objeto\_Original FROM Animales, json\_each(Animales.Identificadores)*

*WHERE Animales.ROWID != NEW.ROWID AND json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND*

*("Salida YN" == 0 OR "Salida YN" IS NULL)*

*AND*

*"FechaHora Registro" == (SELECT MIN("FechaHora Registro") FROM (SELECT DISTINCT UID\_Objeto, "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND Animales.ROWID != NEW.ROWID AND*

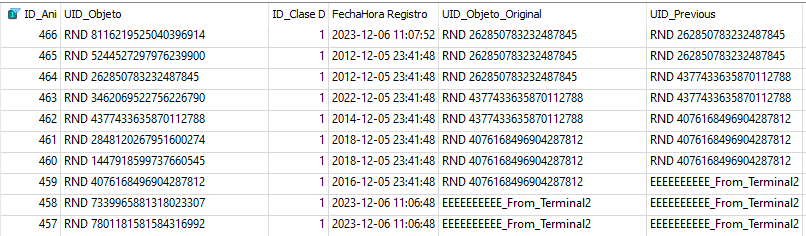
*INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND ("Salida YN" == 0 OR "Salida YN" IS NULL))))*

*WHERE Animales.ROWID == NEW.ROWID;*

*END;*

Two chains of records generated by different Terminals are shown below.

****

****

The problem with this logic is that the Trigger creates multiple chains of records in multiple Terminals that are linked through their “Identifiers” column (they refer to the same physical object), **but they are unrelated** via the UID\_Objeto\_Original / UID\_Previous cols.

In order to link the different chains from different Terminals, “Identificadores” are required.

Now, in reality, each of the lists above is the **duplication of an Object that can be avoided** since it’s done by the same Terminal and the identifiers for the Original Object are set in the first (Original entry) of the object, so any further references to the same identifiers can be detected and prevented.

**All the above exercise is absolutely trivial and unnecessary. It must be resolved in simpler ways:**

* **As part of the entry process, inform the User that an object with those identifiers already exists and provide the options:** 
  + **Cancel:** leave the existing Original Record as the valid one.
  + **Continue:** Use the User provided data as the new data and features of the Object or Entity.

***In this case, UPDATE the existing record with all new values provided by the User.*** *The existing data in the record will be overwritten and lost.*

It’s done this way to preserve the record’s auxiliary fields (ROWID, Duplication\_Index) so that the integrity of the replicated database is maintained at a minimum expense.

**Duplication Index (Definition): an ObjectUID that is stored in the \_Duplication\_Index column as a result of running the duplication check logic that handles replication.**

Duplication stems from the fact that, when nodes are not sync’d the possibility arises that 2 nodes independently create 2 database records to define the same physical object (Tag, Animal, etc.). The Duplication Indices (stored in **\_Duplication\_Index** field in Animal, Tag, Device tables) are list of UIDs used to resolve the duplication problem, which is solved very simply by selecting the record with the earliest fldTimeStamp (the 1st inserted record) as the original record, and this record is used in all operations.

In this way, the problem of Duplication will be limited to matching INDIVIDUAL records from different Terminals (via their “Identifiers”) to detect if they point to the same physical object / entity.

When a match is detected, the logic must flag the Original Record, which is the one among that set with the earliest TIMESTAMP. This will be a search among records from different Terminals (1 record from each Terminal due to replication). In the final state (full replication), the same set of duplicate records will exists in ALL active terminals.

**So how to pull the Original Record**:

* If there are no linked columns (no linking between UID\_Objeto\_Original and UID\_Previous), UPDATING records created by other Terminals is not an issue.
* The Check\_Duplicates Trigger must limit itself to finding the element with the earliest TIMESTAMP among a set of records that share “Identificadores” (the logic for matching Identificadores is already resolved) and setting the field “Duplication\_Index” with the Duplication\_Index of the other records (which will be the same for all records found).
* For reference, “Duplication\_Index” IS a uid.
* The getObject() functions (based on UID or Identifiers values) will ***always return the UID\_Objeto of the record with the earliest TIMESTAMP for the particular value of the “Duplication\_Index”***, with the following logic implemented in python:
  + If UID is passed:
    - Find the UID\_Objecto passed,
    - Find the Duplication\_Index associated to the UID\_Objeto passed.
    - SELECT all records with the Duplication\_Index found, and return the UID\_Objeto for the record that has the MIN(“FechaHora Registro”).
  + If Identifiers are passed:
    - Find the first UID\_Objecto with the Identifier that complies with the criteria selection for Identifiers for the particular db table.
    - Find the Duplication\_Index associated to the UID\_Objeto passed.
    - SELECT all records with the Duplication\_Index found, and return the UID\_Objeto for the record that has the MIN(“FechaHora Registro”).

Simple, right?

The resulting INSERT Trigger is:

**Initial:**

*CREATE TRIGGER IF NOT EXISTS* ***Trigger\_Animales\_INSERT*** *AFTER INSERT ON Animales FOR EACH ROW*

*BEGIN*

*UPDATE Animales SET Terminal\_ID = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1) , Duplication\_Index = NEW.UID\_Objeto WHERE Animales.ROWID == NEW.ROWID AND NEW.Terminal\_ID IS NULL;*

*UPDATE Animales SET Duplication\_Index = (SELECT DISTINCT Duplication\_Index FROM (SELECT DISTINCT Duplication\_Index FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND ("Salida YN" == 0 OR "Salida YN" IS NULL)))*

*WHERE Animales.ROWID == NEW.ROWID;*

*END;*

**Intermediate (NOT using Terminal\_ID. Uses \_Duplication\_Index):**

*CREATE TRIGGER IF NOT EXISTS Trigger\_Animales\_INSERT AFTER INSERT ON Animales FOR EACH ROW*

*BEGIN*

*UPDATE Animales SET Terminal\_ID = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1) , \_Duplication\_Index = NEW.UID\_Objeto*

*WHERE Animales.ROWID == NEW.ROWID AND \_Duplication\_Index IS NULL;*

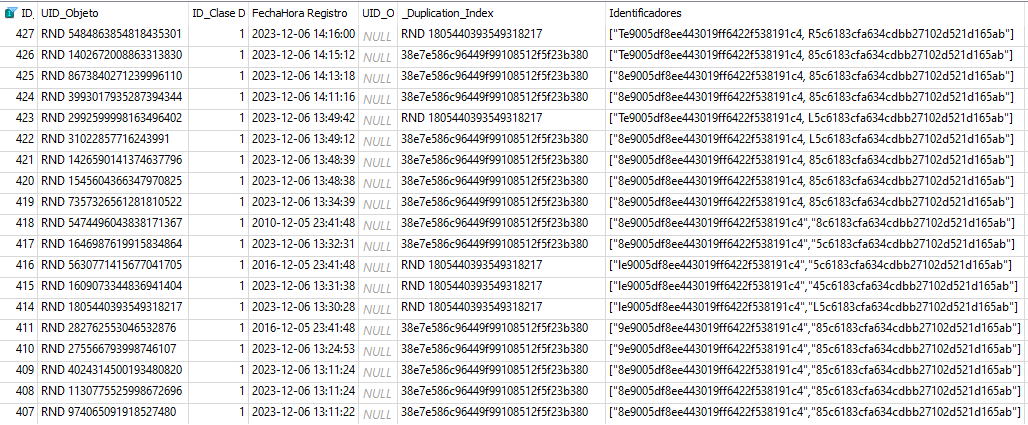
*UPDATE Animales SET \_Duplication\_Index = (SELECT DISTINCT \_Duplication\_Index FROM Animales, json\_each(Animales.Identificadores)*

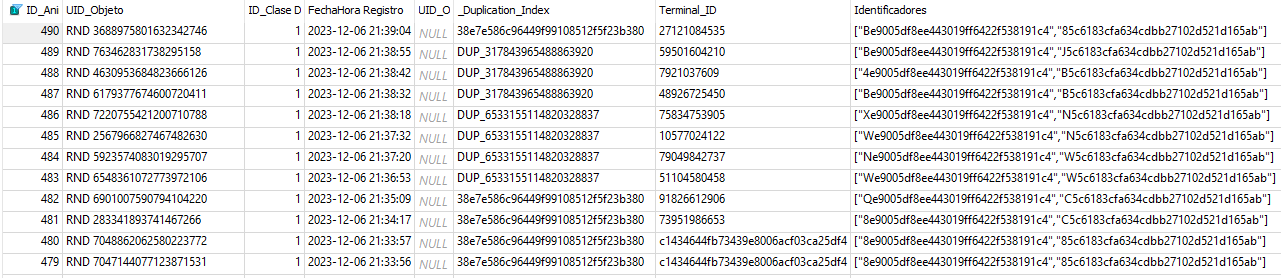
*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND ("Salida YN" == 0 OR "Salida YN" IS NULL))*

*WHERE Animales.ROWID == NEW.ROWID;*

*END;*

And the resulting Data from Animales:

****



**IMPORTANT NOTES on behavior:**

**1) In the case of Animales, Personas, Dispositivos (matching of any 1 of the items in “Identificadores”).**

By design, the Trigger will pull the first record that matches the 1st item listed in “Identificadores”. The logic for handling conflicts in the “Identificadores” list must by handled by python during the assignment of “Identificadores” items to the Object (Tag.assign() / Tag.deassign() methods). The behavior is shown in the table above: see how \_Duplication\_Index behaves as the 1st character in “Identificadores” change from “8” to “L” and “T” at the top list and in line 490 in the bottom list.

In the case of Identifiers that compare as equal (Geo, Caravanas), this behavior does not appear as the number of items in “Identifiers” is always 1.

Another important feature: This trigger no longer uses Terminal\_ID.

Terminal\_ID will continue to be supported in the table; in case it’s needed at a later stage by other processes.

**2)** The Triggers above will generate different values for \_Duplication\_Index for each Terminal where data replication occurs. The value assigned to the list of duplicate records will be the 1st (Initial) value set by the very first record created by the Terminal.

The python logic will work out the correct UID\_Objeto (based on MIN(“FechaHora Registro”)) using the value of \_Duplication\_Index assigned to the particular Terminal.

Now, *THIS CREATE A MULTIPLE REPLICATION LOOP PROBLEM*, which should be solved by assigning

\_Duplication\_Index = UID\_Objeto from the record with that has the MIN(“FechaHora Registro”) at all times.

For this, a logic to process “previous” records will have to be implemented via the processDuplicates() function in python.

In the final stage (all Terminals synchronized), all the \_ Duplication \_Index records will have the UID\_Objeto from the record with MIN(“Fecha\_Hora Registro”). If a record with a new MIN() appears, then all the records in all Terminals will be updated to the new \_Duplication\_Index.

* We can start by leaving \_Duplication\_Index = NULL in all new records, in all Terminals.
* Then, when a duplicate is detected by the Trigger, it sets a value (TBD).
* 2 or more Terminals isolated will generate never generate \_Duplication\_Index values, as \_Duplication is only detected when replication occurs.
* The objective is to recognize, during the replication process, when 2 records share their Identifiers and assign a unique \_Duplication\_Index that in turn can be replicated to the rest of the Terminals without generating infinite looping. -> This in principle could be achieved if the starting point for ALL \_Duplication\_Index is NULL. LET’S SEE. The logic would be:
  + 1) Only values of \_Duplication\_Index == NULL are changed by any given node (Terminal). In this way, the number of modifications of \_Duplication\_Index for any record is limited to 1.

The propagation of duplication (setting of \_Duplication\_Index) is a progressive, cascading process that is triggered when one node receives a replicated record with matching Identifiers to be flagged as Duplicated.

* + **Situation 1**. Initial state -> 2 records with \_Duplication\_Index = NULL: Updated by the node (Terminal) with its own value. ***Trigger runs and updates both records. Trigger must isolate the local record in NULL first****. Since we’re setting the INSERTed record \_Duplication\_Index = UID\_Objeto (which is a unique value across all Terminals) during record creation. Situation 1 is absorbed by Situation 3 by:*
  + *-> setting \_Duplication\_Index = UID\_Objeto during at record creation time (already implemented too) as*

*UPDATE Animales SET Terminal\_ID = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1) , \_Duplication\_Index = NEW.UID\_Objeto*

*WHERE Animales.ROWID == NEW.ROWID AND Terminal\_ID IS NULL;*

***+***

*using the Trigger from Situation 3, implemented as:*

*UPDATE Animales SET \_Duplication\_Index = (SELECT DISTINCT \_Duplication\_Index FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND ("Salida YN" == 0 OR "Salida YN" IS NULL))*

*WHERE Animales.ROWID == NEW.ROWID;*

*END;*

The critical point is to join all records sharing identifiers with a common index. The value of the index is not relevant, only its equality. Eventually, the Original (ie, FIRST) record will be pulled everytime by the GetObject() functions by SELECTing all records with a common \_Duplication\_Index and picking the record with MIN(TIMESTAMP) among all of them.

* + **Situation 2**. Duplication with Received record != NULL: ***This is a no action. No Triggers required*** *since \_Duplication\_Index is set to record’s UID\_Objeto value during record creation****.***

This updated record with then be replicated back to all nodes.

Situations 1 and 2 are simpler from the replication standpoint, as they involve modifying (and replicating) local records (generated by the local node, with \_Duplication\_Index values set during record creation).

* + **Situation 3**. Duplication when INSERTed (replicated) record is found to share “Identificadores”, as defined for the table value. This is straightforward, the trigger is already designed. THIS IS THE ONLY RELEVANT ACTION OF ALL 4 Situations.
    - * See if we can limit the trigger execution to WHEN NEW.\_Duplication\_Index == NEW.UID\_Objeto is TRUE. This should work fine according the logic, but more testing is required.
      * As of 06 dec, this is not possible if we want to pack all UPDATES in just 1 Trigger, which is the case for now…
  + **Situation 4.** Duplication with Local Record doesn’t exist in node (probably not yet replicated). INSERTed record \_Duplication\_Record != NULL: ***Record is stored as is. Nothing else to do.*** When records are created or received via replication with matching Identifiers, the case falls into Situation 3. This is straightforward.

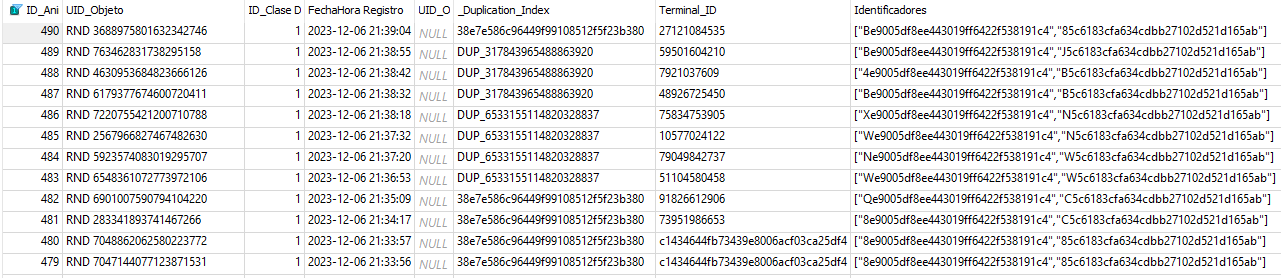
From all this discussion:

1. Trigger must run on ALL INSERTs to the database (Local and Replicated). The trigger logic defines whether the \_Duplication\_Index field in the INSERTed record is or is not modified.
2. Set \_Duplication\_Index = UID\_Objeto during record creation as shown above.
3. Use the already defined trigger for Situation 3 as shown above.
4. This logic should solve the issue of infinite replication recursion loops, as the Triggers defined are **only for INSERTs** and the replication process with updated values will always involve UPDATEs.

*As a corollary, AVOID USING UPDATE Triggers to stay away from replication recursion loops.*

This should cover all the possibilities in replication.

Some results shown below. The 2 NULL values in \_Duplication\_Index is from 2 external terminals that “sent” their records with the column improperly initialized, and as they were non-duplicated records (all new Identifiers), \_Duplication\_Index was (correctly) not assigned. In the real world, these records will be INSERTED with the \_Duplication\_Index properly initialized.



**Final Triggers. End of day 06-Dec-2023.**

Animales (Equivalent for Personas, Dispositivos. Same search using json.each):

*CREATE TRIGGER IF NOT EXISTS Trigger\_Animales\_INSERT AFTER INSERT ON Animales FOR EACH ROW*

*BEGIN*

*UPDATE Animales SET Terminal\_ID = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1) , \_Duplication\_Index = NEW.UID\_Objeto*

*WHERE Animales.ROWID == NEW.ROWID AND \_Duplication\_Index IS NULL;*

*UPDATE Animales SET \_Duplication\_Index = (SELECT DISTINCT \_Duplication\_Index FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND ("Salida YN" == 0 OR "Salida YN" IS NULL))*

*WHERE Animales.ROWID == NEW.ROWID;*

*END;*

Caravanas:

*CREATE TRIGGER IF NOT EXISTS "Trigger\_Caravanas\_INSERT" AFTER INSERT ON "Caravanas" FOR EACH ROW BEGIN*

*UPDATE "Caravanas" SET Terminal\_ID = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1), \_Duplication\_Index = NEW.UID\_Objeto WHERE "Caravanas".ROWID == NEW.ROWID AND \_Duplication\_Index IS NULL;*

*UPDATE "Caravanas" SET '\_Duplication\_Index' = (SELECT DISTINCT \_Duplication\_Index FROM "Caravanas" WHERE Identificadores\_str == NEW. Identificadores\_str AND "FechaHora Registro" == (SELECT MIN("FechaHora Registro") FROM (SELECT DISTINCT "FechaHora Registro" FROM Caravanas WHERE Identificadores\_str == NEW.Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL))) AND ("Salida YN" == 0 OR "Salida YN" IS NULL))*

*WHERE "Caravanas".ROWID IN (SELECT "ID\_Caravana" FROM (SELECT DISTINCT "ID\_Caravana", "FechaHora Registro" FROM "Caravanas" WHERE Identificadores\_str == NEW.Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL) AND "FechaHora Registro" >= (SELECT MIN("FechaHora Registro") FROM (SELECT DISTINCT "FechaHora Registro" FROM "Caravanas" WHERE Identificadores\_str == NEW.Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL)))));*

*UPDATE \_sys\_Trigger\_Tables SET TimeStamp = NEW."FechaHora Registro" WHERE DB\_Table\_Name == "Caravanas" AND \_sys\_Trigger\_Tables.ROWID == \_sys\_Trigger\_Tables.Flag\_ROWID AND NEW.Terminal\_ID IS NOT NULL;*

*END;*

Geo:

*CREATE TRIGGER IF NOT EXISTS Trigger\_Geo\_INSERT AFTER INSERT ON "Geo Entidades" FOR EACH ROW*

*BEGIN*

*UPDATE "Geo Entidades" SET Terminal\_ID = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1) , \_Duplication\_Index = NEW.UID\_Objeto*

*WHERE "Geo Entidades".ROWID == NEW.ROWID AND \_Duplication\_Index IS NULL;*

*UPDATE "Geo Entidades" SET \_Duplication\_Index = IFNULL((SELECT DISTINCT \_Duplication\_Index FROM "Geo Entidades"*

*WHERE Identificadores\_str == NEW. Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL)), NEW.UID\_Objeto)*

*WHERE "Geo Entidades".ROWID == NEW.ROWID;*

*END;*

**07-Dec-2023. Duplicates Management Implementation.**

In order to make the management completely transparent, the duplication logic is implemented at DataTable level, running a routine inside DataTable.\_\_init\_\_() method so that the data returned in DataTables contains only the original records and returned (using the MIN(TIMESTAMP) criterion). The routine strips all duplicates before returning from the DataTable constructor.

The process will be driven by argument *remove\_duplicate\_records*, which defaults to True.

The advanced user can when needed switch this argument to False in order to return all records in the datatable, including duplicate records.

The routine is shown below.

**08-Dec-2023. Duplication Final Form and Definitive Concepts.**

**Definition**: Record Duplication are record-creation events occurring in specific db tables that are linked to physical objects or data entities and that result in multiple, independent records, associated to the same physical object or data entity. They are created independently by different nodes/terminals that run the system.

It stems from the design of the Replication Logic (using Marmot) which has an *eventually consistent* replication approach, and from the fact that the nodes (terminals) running the system are not assumed to be online and connected to each other all of the time.

Record duplication events will arise when one or more terminals work offline and duplication of a record already existing in a Terminal (based on their defined Identifiers) can and must be prevented by the system operation. That is, a Terminal cannot create a duplicate of one of its own records. This case is trivially managed by the system.

It occurs when more than 1 terminal defines (creates) records in the system that refer to the same physical object (Animal, Person, etc.) or entity (Geo Entity).

In this case, each terminal may also generate a number records in other tables, linked to the duplicate record. These linked records are *not* subject to duplication management).

The odds of duplication occurring are relatively low however, if it happens, the risks to database and data integrity and the results produced are disruptive. Therefore, it must be effectively addressed.

**Management**: Record duplication is managed by an SQLITE INSERT trigger defined for each of the tables that support duplication.

* **Detection**: Duplication is detected by the repetition of identifiers associated with the object (Tags for Animals and Devices, Containers for Geo Entities, TagNumbers for Tags).
* **Processing**: Duplication is processed with the aid of an auxiliary column *\_Duplication\_Index* using AFTER INSERT Trigger that updates the *\_Duplication\_Index* value for records that conform to the logic of matching identifiers for the record.

The aim of processing is each node/terminal is to single out 1 specific record as the original record. This original record is returned via API calls to the user so that all related data is linked always to the original record in the node.

* The uniqueness of *Duplication\_Index* is what defines the duplicate items, not its particular value.
* **Earliest Duplicate Record** (updated 13-Dec-2023)**:** Defined on a node-by-node basis. A Duplicate Record in a node is the record with the *earliest value of “fldTimeSamp” field (earliest record TIMESTAMP) that share a \_Duplication\_Index value with other records*. This is to say, the earliest-created record found in the set at the time of routine execution.
  + The Earliest Duplicate Record may or may not meet the condition *UID\_Objeto = \_Duplication Index.*
  + the Earliest Duplicate Record may and will be different in different nodes that are have not completed the replication process.
  + The Earliest Duplicate Record in a node may vary as replication occurs and more matching records are added to the duplicated set.
  + Important property: The Earliest Duplicate Record is assumed to be, at all times, the ***valid*** record of a set of duplicates, meaning this record holds the true and valid data in each field.

As a result, during the process of duplication handling, the field values of the Earliest Duplicate Record will be copied to ALL duplicates records with a later TIMESTAMP in a given node, effectively overwriting their data.

Record field excepted from being overwritten are:

* *UID\_Objeto (fldObjectUID),*
* *“FechaHora Registro” (fldTimeStamp),*
* *Terminal\_ID (fldTerminal\_ID).*
  + sddfsdf
* The ­*\_Duplication\_Index* as defined above (and because of the logic in the INSERT Triggers) is always the Earliest Duplicate Record in a system, at any given time.
* **Original Record** (Updated 13-Dec-2023)**:** The record with the *earliest value of “fldTimeSamp” field (earliest record TIMESTAMP) that share a \_Duplication\_Index value with other records* ***at system (all nodes) level.***

*The Original Record is then the Earliest Duplicate Record with the earliest TIMESTAMP across ALL nodes in a system.*

This record always meets the condition *UID\_Objeto = \_Duplication Index.*

The Original Record for a set of duplicates can only be attained after **total replication consistency has been achieved across all nodes in a system.**

This condition cannot be enforced or waited for in a normal-running system, so all the logic that follows must work with Earliest Duplicate Records in each node. As replication occurs, Earliest Duplicate Records will change in the nodes until, eventually, the Original Record is replicated to all nodes.

* **Corollary**: A record that meets ­*UID\_Objeto = \_Duplication\_Index* is a record that is either the Original Record or the Earliest Duplicate in a node. This is because ­*UID\_Objeto = \_Duplication\_Index* is the initial state for all records in such a table, so an unchanged value means that it remains as the Earliest Duplicate in the node.

If the condition ­*UID\_Objeto == \_Duplication\_Index*. holds then:

1. This is the Earliest Duplicate Record and there are NO duplicates.
2. This is the Earliest Duplicate Record and there ARE duplicates.
3. 🡪 In all cases, the duplication management logic will have to browse the *\_\_active\_duplication\_index\_dict* in order to pull eventual duplicate record uids.

* **Logic**:

**All duplicate records share the same value of \_Duplication\_Index across all nodes**.

**The purpose of the logic twofold:**

1. **Always provide the Original Record data to API calls so that linked records always refer to the Original Record.** But we said that the Original Record can change over time in a given node so it also must
2. **Maintain the integrity of the data (records) associated to all duplicate records so that queries and updates of these associated records remain consistent and maintain integrity in the lifetime of the system.**
   * When a record is replicated onto a node, the logic inspects its identifiers. Upon detection of a duplicate, *\_Duplication\_Index* in the INSERTed record is updated with the *\_Duplication\_Index* value of the record already existing in the receiving node.

Later, this modified record will be replicated back to its originating node with the updated value of *\_Duplication\_Index.*

* + The *\_Duplication\_Index* is initially set upon record creation to the value of the record ***UID\_Objeto***field. This is done by detecting a NULL value in *\_Duplication\_Index.*
  + When a duplicate record arrives in the node/terminal via replication, its *\_Duplication\_Index* is updated by the table Trigger to the *\_Duplication\_Index* value of the already existing record.

Therefore, the *\_Duplication\_Index* value of a set of duplicated records *is the value of \_Duplication\_Index of the* ***1st replication of the record***. This can happen at any time, between any two arbitrary terminals.

* + The API calls handling requests for all tables that support Duplication will, in real time, identify the Original Record and return it to the User. The process is completely transparent to the user.
  + dfsdf
* Setting the *\_Duplication\_Index* value (firing the AFTER INSERT Trigger on a given record) must happen only once, when the 1st replication of that record occurs and the receiving terminal detects the record as a duplicate. NO OTHER ACTIONS SHOULD BE NEEDED: The replication logic in Marmot should take over and broadcast the UPDATE on that record as subsequent replication events occur between pairs of terminals, as they become linked.

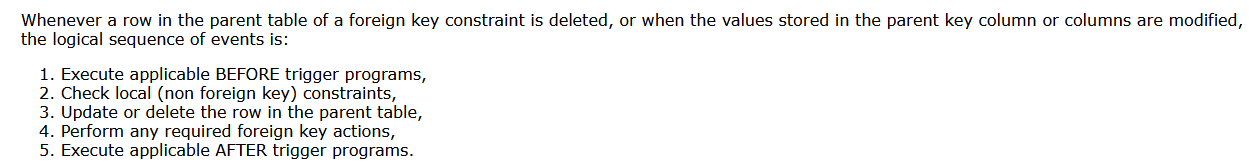
For this to work, we’ll have to go back to selecting the record with the MIN(TIMESTAMP) from the set and UPDATE ONLY IF the received record has a later TIMESTAMP than the MIN found. Otherwise, leave the received record untouched.

The simple logic for all this to work under Marmot Replication and using 1 db Trigger is:

When a record is received in a node/terminal, its ­*\_Duplication\_Index* field will only be updated to a new value if its TIMESTAMP is older than the MIN(TIMESTAMP) of the duplicate records already present in the node/terminal. (The duplicate records are defined as those records with the same *\_Duplication\_Index* value).

Marmot replicator takes care of the rest.

FYI excerpt from SQLITE FK chapter below.



**Local UPDATE. Definition**: UPDATE operation on a duplicate record done by the firing of the Duplicates Mgmt Trigger.

**Repli UPDATE. Definition**: UPDATE operation on a duplicate record generated by a *replication action* on the record. Its an UPDATE of record data generated by another Terminal, that reaches the record on the local terminal via replication.

Repli UPDATEs are of no effect to trigger logic or replication execution, as they are part of an UPDATE operation, and there are no Triggers ON UPDATE.

The resulting trigger is:

*CREATE TRIGGER IF NOT EXISTS* ***Trigger\_Animales\_INSERT*** *AFTER INSERT ON Animales FOR EACH ROW BEGIN*

*UPDATE Animales SET Terminal\_ID = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1), \_Duplication\_Index = NEW.UID\_Objeto*

*WHERE Animales.ROWID == NEW.ROWID AND \_Duplication\_Index IS NULL;*

*UPDATE Animales SET (\_Duplication\_Index) = ((SELECT DISTINCT \_Duplication\_Index FROM (SELECT DISTINCT \_Duplication\_Index, "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND ("Salida YN" == 0 OR "Salida YN" IS NULL))*

*WHERE*

*"FechaHora Registro" == (SELECT MIN("FechaHora Registro") FROM (SELECT DISTINCT "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0*

*AND Animales.ROWID != NEW.ROWID AND ("Salida YN" == 0 OR "Salida YN" IS NULL))))*

*WHERE Animales.ROWID == NEW.ROWID)*

*AND*

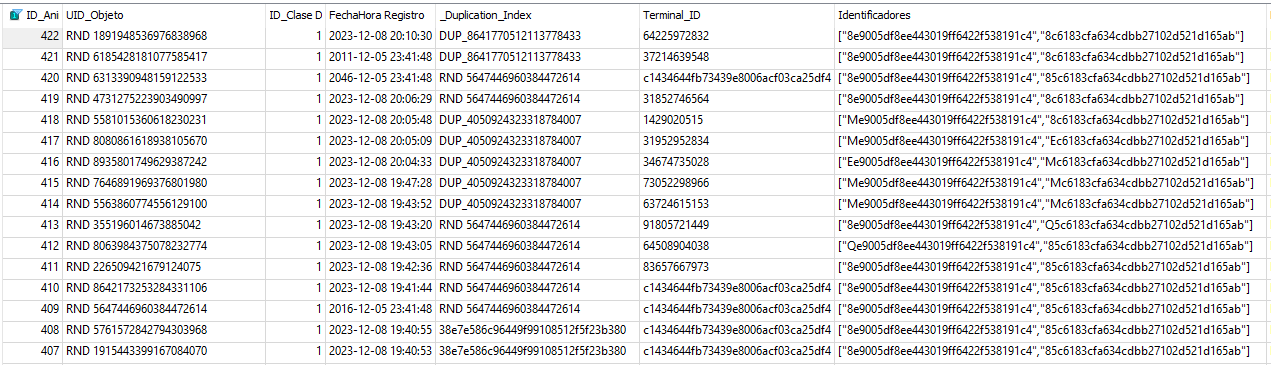
*(SELECT MIN("FechaHora Registro") FROM (SELECT DISTINCT "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0*

*AND Animales.ROWID != NEW.ROWID AND ("Salida YN" == 0 OR "Salida YN" IS NULL))) < NEW."FechaHora Registro";*

*END;*

And the behavior on the table is:



The behavior of the *\_Duplication\_Index* col based on the values in “FechaHora Registro” is consistent with the design criteria established above.

The Trigger explained is:

*CREATE TRIGGER IF NOT EXISTS* ***Trigger\_Animales\_INSERT*** *AFTER INSERT ON Animales FOR EACH ROW BEGIN*

*UPDATE Animales SET Terminal\_ID = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1), \_Duplication\_Index = NEW.UID\_Objeto*

*WHERE Animales.ROWID == NEW.ROWID AND \_Duplication\_Index IS NULL; 🡪* Sets values for Terminal\_ID and \_Duplication\_Index = UID\_Objeto ONLY IF the value of \_Duplication\_Index is NULL. This line is executed only once in the life of the record, during its creation in the “mother” terminal.

The importance of using UID\_Objeto is because it’s a UNIQUE value across all terminals (no 2 repeat UID\_Objeto in any of the terminals, at any point). And because it’s linked directly to the record UID and hence to the underlying physical objects described by the record.

When queries are performed on the objects, the value in column *\_Duplication\_Index* is always returned as the value of the “Original” record.

*UPDATE Animales SET ( \_Duplication\_Index) = (SELECT DISTINCT \_Duplication\_Index FROM (SELECT DISTINCT \_Duplication\_Index, "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND ("Salida YN" == 0 OR "Salida YN" IS NULL))*

*WHERE*

*"FechaHora Registro" == (SELECT MIN("FechaHora Registro") FROM (SELECT DISTINCT "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0*

*AND Animales.ROWID != NEW.ROWID AND ("Salida YN" == 0 OR "Salida YN" IS NULL))))*

*WHERE Animales.ROWID == NEW.ROWID*

*AND*

*(SELECT MIN("FechaHora Registro") FROM (SELECT DISTINCT "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0*

*AND Animales.ROWID != NEW.ROWID AND ("Salida YN" == 0 OR "Salida YN" IS NULL))) < NEW."FechaHora Registro";*

*END;*

1. Do a search of all records in the table, excluding the NEW record just inserted (this is done with *AND Animales.ROWID != NEW.ROWID*) that share one or more identfiers. This is done using the json functions: *SELECT DISTINCT "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores) WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0 🡪* the test is done using a string function (INSTR), which requires of a definition of an auxiliary column in the table: *Identificadores\_str****,*** which is a generated column. Totally transparent to users.
2. From list of records obtained in 1), if any, get the MIN value of “FechaHora Registro” 🡪 *SELECT MIN("FechaHora Registro")*
3. And compare that MIN value with the TimeStamp of the newly INSERTed record 🡪 *< NEW."FechaHora Registro";*
4. If the comparison in 3 is False, the UPDATE of *\_Duplication\_Index* is NOT executed. The field is left as is.
5. IF the comparison is True (MIN() is less than *NEW."FechaHora Registro"*), then set *\_Duplication\_Index of the record just INSERTed = \_Duplication\_Index value of the record just found through the MIN() operation:*

*UPDATE Animales SET \_Duplication\_Index = (SELECT DISTINCT \_Duplication\_Index FROM (SELECT DISTINCT \_Duplication\_Index, "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND ("Salida YN" == 0 OR "Salida YN" IS NULL))*

*WHERE*

*"FechaHora Registro" == (SELECT MIN("FechaHora Registro") FROM (SELECT DISTINCT "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0*

*AND Animales.ROWID != NEW.ROWID AND ("Salida YN" == 0 OR "Salida YN" IS NULL))))*

*WHERE Animales.ROWID == NEW.ROWID*

1. For the other tables (Caravanas, Geo, Dispositivos, Personas) the logic is exactly the same. Only varies the validation conditions of the Identifiers (they are simpler, in fact, as the json functions are not needed. The comparison is done over a single value per record as opposed to a JSON List of values as is the case with Animales).
2. Simple. Right?

**09-Dec-2023.**

One last addition to the trigger above: When fired, besides updating the *\_Duplication\_Index* field, will *also UPDATE ALL remaining fields in the row,* ***except for fldID, fldObjectUID fields and fldTimeStamp. Let’s see how this goes with the Replication process.*** This is to ensure consistency of the data and enable the operation according to the New Paradigm described below.

With the current implementation of the Trigger, updating the fldTimeStamp field will only occur when the condition

*(SELECT MIN("FechaHora Registro") FROM (SELECT DISTINCT "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0*

*AND Animales.ROWID != NEW.ROWID AND ("Salida YN" == 0 OR "Salida YN" IS NULL)))* ***< NEW."FechaHora Registro";***

is met, that is, all fields (including fldTimeStamp) are updated in a replicated node only if the incoming record’s TimeStamp is later than the TimeStamps of the duplicate records already existing in the node.

18-Jun-2024: Why is all this above of 09-Dec necessary??? The incoming record has already all of its fields properly populated. The only 1 field that needs update is \_Duplication\_Index.

This means that in subsequent replications of the updated record, *UPDATEd timestamps can only be earlier than the initial TimeStamp* they first were sent out from the sending terminal 🡪 an UPDATE operation will ensue to UPDATE the fields of the record in the sending node.

*Since UPDATEs have no triggers at the time, the change in fldTimeStamp (or any other field in the record for that matter) will have no effect on the trigger execution or replication process. However, note that UPDATEs are made during replication.*

***UPDATE Triggers should be avoided for simplicity, but if at all needed, must be design with consideration of this situation.***

Use the column-list expression for UPDATE, as in

UPDATE Table SET (\_Duplication\_Index, col2, col3, … col n) = (val1, val2, …, val n) WHERE ROWID=…, etc.

(Beginning in SQLite [version 3.15.0](https://www.sqlite.org/releaselog/3_15_0.html) (2016-10-14), an assignment in the SET clause can be a parenthesized list of column names on the left and a row value of the same size on the right.)

**New Paradigm: Multi-record objects for Duplicates, with 1 Original Record.**

The concept is to detect duplicates and flag them via *\_Duplication\_Index*as described in the previous section and allow all references to those records without interference: create activities, programmed Activities, etc.

Just that when information has to be pulled for the object, the information has to be consistent. This brings the need of data consistency across duplicate records.

So, criteria ***must*** be defined to identify an Original Record and when duplicate records are detected, all their respective fields are made equal (UPDATED) to those of the Original Record (except for fldObjectUID field). For now, the criterion is using TIMESTAMP: The Original Record is the one that was created first, among all terminals. The current Duplication management logic works with this premise.

This way, records are duplicated but all with the same data for queries.

***The objective is that accessing any of the duplicate records, the same data is retrieved at all times.***

18-Jun-2024: Additional criteria has been added to simplify Duplication Management and do better input error detection and management: use of the ***Physically Attached*** concept for Tags. This is fully explained in a separate document: Tag lifecycle, Tag Identifiers implementation>

Triggers will have to be defined for UPDATE and DELETE, so that when an Original Record is UPDATED or DELETED, the change is cascaded to all the Duplicate Records present in that Terminal.

The UPDATE, DELETE actions should work and propagate exactly in the same fashion as when UPDATING *\_Duplication\_Index*. It’s the same concept, using the same criteria to modify records.

**Caution**: the INSERT Trigger executes UPDATEs. Any AFTER UPDATE Triggers will have to use proper WHEN clauses to exclude re-entry and recursion of the Trigger code. Such WHEN clause could use Terminal\_ID of the record to identify its terminal of origin.

However, re-entrant triggers will have to address multi-threading as they may be interrupted during execution and another instance of the trigger being executed by another thread. This would involve parameter saving/switching*: too complicated (and expensive in terms of performance) to implement in SQLite.*

Using the *\_Duplication\_Index* field, the system can process them accordingly when taking counts and other activities.

The base concept is that duplicate records *available at a given time in the Terminal database* will share the same record data, field by field, except for the *fldObjectUID* field (and fldID, of course). Then all applications and functions can access table data without any particular requirements to perform any activity required.

*Available at a given time in the Terminal database* mentioned above means that the number of duplicated records and the data values recorded on them will change over time as replication occurs.

The process stops only when full replication consistency is reached across al terminals: Then all duplicate records are available in all Terminals and no more duplication is possible, as Terminals will detect (and MUST PREVENT) any attempt to create a duplicate element.

*The objective of the logic is that data integrity is maintained as the replication process eventually changes the data on the duplicate records present in each Terminal.*

As a result of this approach, when a query based on Identifiers is performed on a table with Duplicates, n rows (with n > 1) will be produced that match the Identifiers provided and refer to ***the same*** object or entity.

The API must adopt criteria to handle the query: return 1 specific row or return all rows available.

Functions/methods will have 1 of 2 categories:

* **Duplication-indifferent:** Functions/methods that operate regardless of the records being duplicates or not.
* **Duplication -aware:** They need to manage the Duplication status of the records being processed in order to produce consistent output. Ex.: countAnimals(), getObject(). The criteria to handle duplicate records may have to be devised on a case-by-case basis.

With the current approach, the triggers are specific to the db Tables they are defined for (sorting multiple vs. single identifiers, etc. as seen in Animales, Caravanas, Geo, etc.)

These nuances and details are handled by the SQLite Triggers and are transparent to the python interface.

With this, the resolution of duplicates by the python code is reduced to finding set or sets of *fldObjectUID* values that share the same *\_Duplication\_Index* value. 🡪 This can be solved by a single function that receives *fldObjectUID* value or values and returns a set of *fldObjectUID* values related to all the *\_Duplication\_Index* found.

**10-Dec-2023. Improving the AFTER INSERT Trigger for Duplicates handling.**

Try:

1. Eliminate the first UPDATE.

*UPDATE Animales SET Terminal\_ID = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1), \_Duplication\_Index = NEW.UID\_Objeto WHERE Animales.ROWID == NEW.ROWID AND \_Duplication\_Index IS NULL;*

Instead, initialize *Terminal\_ID* and *\_Duplication\_Index* in the python program, during the creation of the Animal record.

NOTE: Setting *\_Duplication\_Index = fldObjectUID* **is critical** to the correct operation of the duplication management logic since the value in *\_Duplication\_Index* is overwritten in every duplicate record (that arrives via replication) with a fldTimeStamp later than the local record.

1. This leaves the 2nd UPDATE to execute ONLY as a Replication INSERT operation, that is, the Trigger fires only on all INSERT operations but ***runs only when the condition in WHEN resolves to True.*** like this one:

*WHEN*

*(SELECT DISTINCT UID\_Objeto FROM Animales, json\_each(Animales.Identificadores)*

*WHERE Animales.ROWID != NEW.ROWID AND json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND*

*("Salida YN" == 0 OR "Salida YN" IS NULL))*

*BEGIN*

This allows for higher efficiency as the trigger fires only when a set of matching Identifiers is found in the table.

UPDATE: Tested. There is virtually no difference between the 2 trigger approaches. 10 – 20 msec execution time.

**11-Dec-2023. Evolution of the AFTER INSERT Trigger**

The existing triggers UPDATEs only the incoming row, using the condition ROWID == NEW.ROWID. This limits the updates on INSERTs to just that row, leaving all other duplicated records (records that share the same identifiers) untouched and therefore, unmodified.

This should be resolved in one operation and with the same efficiency cost by removing the ROWID == NEW.ROWID condition. Let’s see.

This should result in an immediate UPDATE of all duplicate rows in a given terminal as a new duplicate with an earlier TIMESTAMP arrives. With this, the integrity of all records should be sustained at all times.

A ***very useful*** byproduct of this approach is that an improved, faster check for duplicates can be done by checking *fldObjectUID == fld\_Duplication\_Index* for a record:

1. If equal, the record may or may not have duplicates (not yet replicated onto this node), but the current ROWID is the first (oldest) one of all duplicates present in the terminal. Any operations and Activities should be performed on this record.
2. If not equal, then a search for the earliest duplicate must be performed in order to use the correct row for operations on the record.

This logic greatly improves the efficiency as more that 99% of the time it is expected that *fldObjectUID = fld\_Duplication\_Index* and no further searches for duplicate records are required.

Because of the fact that the trigger writes potentially to ALL duplicate records (existing records and also the NEW incoming record), **BEFORE INSERT Triggers cannot be used on the Objects tblas** (as the later INSERT operation could overwrite any values written to the new record by the trigger). However, they can be used for other tables with no problems (ex.: the \_sys\_Terminals table), as long as the data written to the other tables will not be modified immediately by the AFTER INSERT trigger: important limitation that leaves BEFORE triggers with limited application in real life.

This is the trigger, and it finally works seems to work updating all records with newer TIMESTAMPS (later than the min(TIMESTAMP) found among all records:

*CREATE TRIGGER IF NOT EXISTS* ***Trigger\_Animales\_INSERT*** *AFTER INSERT ON Animales FOR EACH ROW*

*BEGIN*

*UPDATE Animales SET Terminal\_ID = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1), \_Duplication\_Index = NEW.UID\_Objeto*

*WHERE Animales.ROWID == NEW.ROWID AND \_Duplication\_Index IS NULL;*

*UPDATE Animales SET (\_Duplication\_Index) = ((SELECT DISTINCT \_Duplication\_Index FROM (SELECT DISTINCT \_Duplication\_Index, "FechaHora Registro"*

*FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND ("Salida YN" == 0 OR "Salida YN" IS NULL))*

*WHERE*

*"FechaHora Registro" == (SELECT MIN("FechaHora Registro") FROM (SELECT DISTINCT "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0*

*AND ("Salida YN" == 0 OR "Salida YN" IS NULL)))))*

*WHERE*

*Animales.ROWID IN (SELECT "ID\_Animal" FROM (SELECT DISTINCT "ID\_Animal", "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0*

*AND ("Salida YN" == 0 OR "Salida YN" IS NULL)*

*AND "FechaHora Registro" > (SELECT MIN("FechaHora Registro") FROM (SELECT DISTINCT "FechaHora Registro" FROM Animales, json\_each(Animales.Identificadores)*

*WHERE json\_valid(Animales.Identificadores) AND INSTR(NEW.Identificadores\_str, json\_each.value) > 0 AND ("Salida YN" == 0 OR "Salida YN" IS NULL)))));*

*END;*

And it works as follows:

Writes multiple records on each INSERT by:

1) Scans all records in the Table that qualify as duplicate records (share the criteria for equal Identifiers).

2) Gets min(TIMESTAMP) among all those records (both the pre-existing or "old" records and the "new" incoming record). The values in the record that has min(TIMESTAMP) will be overwritten to the rest of the duplicate records.

3) Writes data to all records (old and new) that comply with:

a. Their ROWIDs qualify as duplicates (meet the criteria for Identifiers),

b. They are not inactive records ("Salida YN" is 0 or NULL),

c. Their "FechaHora Registro" > min("FechaHora Registro") just found. This prevents from overwriting the record that holds that minimum value. Just a detail.

With this trigger, the byproduct mentioned above comes into effect and the python functions can use it to improve query response time.

**12-Dec-2023. Management of Duplicates. Evolution on the Trigger logic.**

The purpose of this update is driven by the fact that:

Duplicate-aware tables (Animales, Caravanas, Geo, etc) spend a very small amount of time creating and updating records, and most of the time using the data just for query.

With this in mind, the logic must be designed so that all the under-the-hood updates to *\_Duplication\_Index and \_Terminal\_ID* are done by SQLite. Then Python creates and updates memory data structures to access the records in these tables with the minimum number of db queries possible.

* Important: The \_*Duplication\_Index* in the current structure is also the record that **holds the min(TIMESTAMP) at any time**, in a given node, given how a minimum found is overwritten to all existing records sharing a *\_Duplication\_Index.*
* With this, a logic to handle duplicate records and Original Record will be simpler and more efficient to implement, in particular, using memory, as follows:
  + Implement a memory dictionary

*\_\_object\_uids = {\_Duplication\_Index: set of linked uids, including \_Duplication\_Index, }*

* + Here *Dupllication\_Index* is the uid that has the lowest TIMESTAMP**. It is the Original Records at all times.**
  + **To get the Original Record:** Return *\_Dupllication\_Index* (dict key). If the search is via a different uid, it may be needed to sift through the dictionary until the search uid is found, then pull the key for assigned to it.
  + **To get all duplicates:**
    - Find the *\_Dupllication\_Index* key in the dictionary:
      * If found, return a tuple the value associated to it (1 or more records).
      * If not found, sift through the dict values until the uid is found, and return the value (set) it belongs to. Return as a tuple (for ease of processing in SQLite).
* Use a *third* INSERT statement in the AFTER UPDATE Trigger to UPDATE the \_sys\_Terminals table with the dict

{UID\_Objeto: (\_Duplication\_Index, “FechaHora Registro”}

The trigger fires AFTER incoming duplicates are INSERTEed in the terminal and their fields updated by previous UPDATE statements in the trigger, using a WHERE *\_Duplication\_Index <> NULL* so that only records replicated from other terminals modify the \_sys\_Terminals table.

* Then, in python, a background function will read from \_sysTerminals table and update the dictionary \_\_object\_uids in memory.

Simple, uh?

**Memory handling of tables that support Duplication.**

Animales, Caravanas, Personas, Geo, Dispositivos will have 2 specific dicts in memory, as part of the class structure that will be initialized by a \_init\_uid\_dict() function

active\_uids\_dict = {uid: \_Duplication\_Index}

active\_duplication\_index\_dict = {\_Duplication\_Index: set of uids, }

*These 2 dicts defined at Animal subclasses (Bovine, Caprine, etc.), to split them among smaller dicts and improve efficiency of searches.*

These dicts will be updated regularly by a background function: when a duplicate enters the node via replication, SQLite will set a flag in table [\_sys\_Terminals] indicating that the uid/\_Duplication\_Index structure has changed.

* When the flag is detected, the init\_uid\_dict() function will be re-run by the background thread that will update both dictionaries to their latest state.

Queries are by far the most common activity on the db for these tables, so it’s imperative that they are as efficient as possible. With the use of these 2 memory tables, the database queries required to obtain a uid or a set of uids of Duplicate Records is reduced to its minimum.

*Now, we can also make* ***active\_duplication\_index\_dict contain only duplicate items****. In this way, an empty active\_duplication\_index\_dict will mean that there’s no need to look for duplicates: there are none.*

Then, to query:

1. Search is done on the full \_uids\_dict. ALL UIDs of the system will be found there. A *\_Duplication\_Index* is returned as the dict value (Keep the original uid used as key, as it may have to be returned most of the time as the result of this search).
2. With the *\_Duplication\_Index* found, search the \_duplication\_index\_dict:
   1. If *\_Duplication\_Index* is present in \_duplication\_index\_dict, return its value (a list of duplicate uids associated to the I*\_Duplication\_Index*).
   2. If If *\_Duplication\_Index* is not found in the \_duplication\_index\_dict 🡪 The record has no duplicates: return the initial If *\_Duplication\_Index* obtained in 1, which is always the Earliest Duplicate Record.

**16-Dec-2023. On the logic for the Replication Process Management. Some basic thoughts.**

1. Processing replication demands discrimination between *local records* and *replicated records.*
   1. *Local Records:* records generated by the local node.
   2. *Replicated Records:* Records of the same table, generated by other nodes/terminals.

To enable this discrimination, the *Terminal\_ID* field is used in all table that must handle replication.

The *Terminal\_ID* value is created by an SQLITE Trigger when the record is first inserted in the table**. It must be present in ALL table that require replication handling**.

1. What Replication Handling does:
   1. **Object Tables (Animales, Caravanas, Personas, Geo, Dispositivos):** determines whether a record (generated either locally or externally) is duplicated with 1 or more records already existing in the local db.

Based on that, overwrites the ­*\_Duplication\_Index* field of the record based on the TIMESTAMP of all records and defines the “Earliest Duplicate Record” for the node, which is then defined as the “true” record for that node.

It also overwrites all remaining fields of all “newer” records with the data from the Earliest Duplicate.

* 1. sdfsdf

1. **RA (Registro De Actividades) Tables**: Scans all the externally-generated records to determine whether any of them correspond to the closing of a Programmed Activity defined the local node and closes the ProgActivity Records accordingly.

For this, a logic with records’ TimeStamp values is used, assuming that when replication occurs ***multiple records from 1 or several nodes*** will enter the local node. The value written on TimeStamp field will be that of the last record to be replicated.

When the processRA() method is run it will, by design, have to traverse ALL the valid records in the RA Table and identify those that comply with:

* + - Are external (coming from a node with a different Terminal\_ID signature).
    - Their TimeStamp is later than the TimeStamp of the last execution of the function. This field is defined in the ***\_sys\_Triggers\_Table*** table.

The records that meet both criteria are processed by processRA().

1. **RAP (Registro De Actividades Programadas):** We’ll see what’s needed here…
2. Adas

Sdfsdf

**03-Apr-2024. Replication using Marmot.**

**Essentials: Adding a new node.**

Doing replication with Marmot app (leaderless, eventually consistent replication) will require the following in order to add and setup a new device (with its new database) to the system:

* + 1. The database must have the **exact same structure** as the databases existing in the system (which in turn must share that same structure: table names, fields in each table and field names).
    2. The database of the new Terminal / node is assumed **EMPTY** except for defined fields (like Terminal\_ID in \_sys\_terminals table).
    3. One pre-existing database from another node will have to be copied to the new (empty) database. For this, **Data is to be INSERTED. NOTHING can be overwritten** in the new database. To accomplish this:
       1. All active nodes must be made aware of the existence of the new Terminal / node so that any changes performed on other nodes are immediately replicated to the new one, while its database update procedure is ongoing.

To this end, Marmot must initially be setup with flag ***replicate = True.***

* + - 1. The new node MUST NOT perform data entry to the database during the setup process. Access to data entry must be disabled during this time. ***This limitation required while the database update process is ongoing, so that the data copied to the new database is NOT replicated further.***

***Because of this, the source node must set replication = False.***

To this end, Marmot must initially be setup with flag ***publish = False.***

***The same limitation applies to the node picked to provide the data: MUST NOT allow for entry of new data*** in order not to confuse existing data with new data during the data transfer.

* + - 1. A function will pick a preferred node from all the nodes online (probably the one with the largest DB size, assuming this points to the DB with the greatest amount of data).

The user should be required to pick one from a list: one that ensures that will remain online and connected during the whole database update process.

* + - 1. Dedicated application code will perform data requests and transmission between sending and receiving nodes and update the database tables in the receiving node, until completion.

The best way to go about this to prevent duplication of rows during the update, due to incoming replicated data, is to perform a local copy of the database in the picked node and work with that static copy. This way, any replicated data is not added to this database and is added to the db in the new node directly.

* + - 1. Once completed, the database in the new node is in operational condition, fully integrated to the system.

Set Marmot flag ***publish = True*** in order to enable broadcasting changes to this database to the rest of the nodes in the system.

**The rest of this chapter is deprecated. All the code generated will be removed.**

~~The current model for the database is:~~

* ~~1 multiclient SERVER running centrally and supporting all operating devices. The server runs Postgre or similar. The main purposes are:~~
  + ~~Centralize all the DB data collected by the operating devices.~~
  + ~~Server as the focal point for synchronization of data across all devices.~~
  + ~~Backup~~
* ~~Operating Devices (mobiles, tables, pc, etc): Run a fully working version of the DB locally (SQLite) and synchronize with the Server via a defined protocol.~~

~~DESIGN CRITERIA: In order to accommodate the multiple devices and the Server and keep data integrity and simple to manage, DB Indices are assigned in batches by the Server to each device.~~

~~The criteria for batch size include maximum expected total number of devices in the system and local storage of each device, as a minimum.~~

~~The upper limit to index number is given by the upper limit for indices that SQLite handles.~~

**~~03-Jun-23:~~**

~~Data synchronization consists in uploading data from all databases / devices in the system onto a Database Server.~~

~~Many devices may be running the system concurrently, modifying system data on their local copies of the database, with or without an internet link to synchronize data.~~

*~~Devices can be any type of computing element (computer, mobile, tablet, etc.) that has an instance of the system running on it.~~*

***~~There may be more than 1 database assigned to 1 device~~****~~. This is the case of DataStreams, which must be handled with separate databases.~~*

~~The paradigm, for simplicity and efficiency, is having each Database operate with blocks of assigned index numbers, that are unique to each database.~~

~~With that, all databases can be uploaded directly onto the Server or written onto other devices without any DB index conflicts.~~

* *~~The system is designed to operate fully while offline and synchronize when a link (internet, wifi, Bluetooth) is established.~~*
* *~~Synchronization is done at database record level: Each record generated in a database can be managed individually for upload and synchronization. Dedicated fields are present in each record to manage upload/synchronization.~~*

~~There are independent by interrelated steps to data synchronization:~~

1. **~~Data Upload:~~** ~~Data is sent from a local data base to the Server. Done via internet link.~~

~~The server stores blocks of records corresponding to each database operating in the system (again, 1 database is associated to 1 device for now).~~

1. **~~Data Sync:~~**
   1. **~~Server Sync:~~** ~~Once a record is successfully uploaded:~~
2. ~~The server acknowledges the reception to the sending devices by sending it a message.~~
3. ~~In turn, the device updates its internal record to flag it as uploaded.~~
4. ~~The server (that holds a list of all active databases) appends the record to an internal structure that specifies all the devices that the record must be sent to.~~

~~On their side, the devices will~~ ***~~pull~~*** ~~the records kept for them with a background thread that will ping the server whenever a connection is available.~~

~~When the record is pull into the device, the device will update its local database to reflect that the record has been sync’d (set the sync bit to 0).~~

1. ~~The device will send a handshake to the server to indicate that the record has been received.~~
2. ~~When the server gets the handshake, it will:~~
   1. ~~Remove the record from the data structure corresponding to the device that did the sync’ing.~~
   2. ~~Update the record in its own database as sync’d (sync bit = 0)~~

*~~Important note: Client must pull the records meant for them from the Server. This is for efficiency purposes, as the resources are used only when a connection is available on the client side, freeing the server from duties of pinging constantly all clients for a connection.~~*

*~~In this sense, the server operates as an~~* ***~~always-on repository~~*** *~~for clients to send their data to and pull (fetch) data that’s stored for them in the server.~~*

* 1. **~~Local Sync (serverless):~~** ~~When a server is not available but 2 or more devices are connected locally (bluetooth, wifi, etc.), the synchronization process can be performed among the locally connected devices.~~

1. ~~Each device will run through its list of records to sync and will send those records to the other devices.~~

~~ii. Upon reception, the other devices will store the records and send back a completion message to the sending device.~~

~~iii. The sending device will update an internal list (json list) with the names of the devices that have been synchronized. This list will be later used by the server (when the records are uploaded) for the server to determine which devices already have those records and then not re-send that data to them.~~

**~~Index Assignment:~~**

~~The system can operate in 2 ways:~~

1. ~~With database sync.~~
2. ~~Without database sync: allows for specific cases when the system will run in 1 device and 1 device only.~~

~~The mode is established during system setup and cannot be changed (for now??)~~

~~If the system runs~~ *~~with database sync~~* ~~the assignment of lower and upper indices to each device is to be done (at this time) by inserting the data in a dedicated table in the Database for each device. Then, the system can establish the indices limits at initialization time.~~

~~The indices are assigned then by a low-level kernel function, \_\_get\_max\_id (), when the db-write functions are called.~~

~~When the upper limit of indices is reached, the system must provide another block of indices for the DB to continue to operate.~~

**~~Synchronization Fields~~**~~: Each record in the database has the following fields.~~

1. ~~Bitmask: INTEGER to be treated as bits, for multiple record status. At the time being:~~

~~Bit0: Upload Bit~~

~~Bit1: Sync Bit.~~

~~Hence 1 means the record is to be uploaded on server (backup purposes) but not synchronized across devices. This is the case for records in \_sys table (\_sys\_Tables, \_sys\_Fields, \_sys\_Time\_Reference)~~

~~2 means the record is to be sync’d across devices BUT NOT uploaded to server.~~ **~~This option is not used at the moment.~~**

~~3 means the record is to be uploaded~~ **~~and~~** ~~sync’d across devices.~~

1. ~~PushUpload (JSON) -> to be renamed to “syncd Devices”: JSON list of the devices the record has already been sync’d with, in order to avoid sending duplicate data over a link (internet, etc).~~
2. ~~TimeStamp Sync: TIMESTAMP field with the date of synchronization: allows to manage record synchronization by their time signature.~~

~~fldBitmask is used in setRecord, setRecords, delRecord. If it’s not loaded from DB, the default bitmask for the table is assumed (default bitmask can be obtained with a call to the getTblName () function.~~

~~The background thread that saves data to DB and uploads to the server must make a DB read to pull the~~ *~~Syncd List~~* ~~field (PushUpload field at the time of this writing) if not yet part of the record, and append it to the record. This data must be passed to the server for it to determine the databases it must present the record to, in order to complete synchronization of the record.~~

~~The background thread must also append the~~ *~~TimeStamp Sync~~* ~~record (if not present) and insert the timestamp. In this case no database reads are required.~~

**~~Non-sync data~~**~~: In general, all records produced internally by the system (category changes, timeout due to passage of time, maybe notifications, programmed activities generated by the system) should be uploaded to the server (backup) but not sync’d across devices (since the other devices will generate the same records as time passes).~~

~~Upload Bit = 1~~

~~Sync Bit = 0.~~

~~If necessary, this can be individually managed for specific records.~~

**~~Duplicate Records:~~** ~~There are several scenarios for the generation of duplicate or at least equivalent/redundant records:~~

1. ~~2 (or more) devices, independently create an Activity record (Inventory, Status, Destete, TM, etc).~~
2. ~~2 (or more) devices independently create a programmed activity (progActivity).~~

~~The logic will be as follows:~~

1. ~~Executed/Performed Activities: All records are considered valid and uploaded. The current record will be the last one executed (pulling~~ ***~~fldDate field~~*** ~~from Activities Data Table) -> Important, do not confuse this date with the TimeStamp field in Registro De Actividades.~~
2. ~~progActivities:~~
   1. ~~progActivities generated by the system are uploaded (backed up on server) but not sync’d across devices.~~
   2. ~~All progActivities assigned to an object are uploaded and registered with that object during system startup.~~
   3. ~~All user-generated progActivities are considered valid and are assigned to the required objects.~~
   4. ~~When checking for closure, the system will close all progActivities meeting the conditions. This will close any equivalent or redundant progActivities for a given target object (Animal, Person, etc).~~
   5. ~~At closure time, the system will also check and close all progActivities meeting the conditions and with fldProgrammedDate expired.~~

~~The closure of a progActivity for an individual object is done by setting the fldClosingActivity field to the closing activity fldID field in tbl LinkPA.~~

* 1. ~~Finally, the checkFinalClose() method is run for all progActivities found.~~

**~~Block TimeStamp~~**~~: A timestamp set by the background thread that sets a record for upload. When the thread stores the record in database, making it ready for transmission, it sets the timestamp (using time\_mt) to all records in the block. Later on, that record will be transmitted, once communication with the server is established. Block timestamp can be used to sort and prioritize records for transmission.~~

~~See pages S6 – S10 in handwritten notes.~~

**~~5-Jun-23:~~**

**~~TODO:~~** ~~The background thread that uploads records to the server and downloads sync data for the device will have to:~~

1. ~~Append the Device ID to each record to identify for the server which device originated the record.~~
2. ~~Implement a logic to pack records in packet sizes in line with the system performance / link speeds in order to maximize efficiency of the transmission for link conditions.~~
3. ***~~The communications threads will not be background~~***~~. The threads will run an infinite loop pinging the server and pinging other active devices, and uploading and sync’ing data when server or devices become available.~~

**~~17-Jun-2023~~**~~:~~

**~~INSERT / UPDATE / DELETE records logic~~**~~.~~

~~The server will have to implement a check for UPDATE records:~~

1. ~~If a record received for UPDATE is not found in the Server DB, then the original INSERT was never received. Then the operation becomes an INSERT.~~
2. ~~With the above, if a record is received for INSERT but already exists, it will trigger an error on INSERT -> the creation is ignored.~~
3. ~~If a record received for UPDATE has an earlier~~ *~~creation date~~* ~~than the last version of that same record (same fldID) already stored in server, the UPDATE is ignored.~~

~~Still, to be noted is that the scenarios in 1, 2, 3 should not happen: the logic in each device must be careful to send the records in the \_Upload\_To\_Server tables in a sequential, chronological order so that the INSERTS always happen before the UPDATES or DELETES.~~

1. ~~If a DELETE is sent for a record already deleted (not present in the server) the DELETE fails as Record Not Found (and can also be ignored).~~

~~These are simple solutions to a rather complex synchronization problem.~~

**~~Upload/Sync Implementation details:~~**

1. ~~As of now, 2 threads to be implemented as communications threads:~~

* ~~A~~ *~~ServerLink Thread:~~* ~~In charge of all communications with the server.~~
* *~~A DevicesLink Thread~~*~~: Communicating with all devices sending/receiving data~~
* ~~These are regular threads (they don’t need to be TimeInterval threads).~~

1. ~~The ServerLink Thread will:~~
   1. ~~Ping the Server continuously for a connection.~~
   2. ~~Once a connection is established:~~
      1. ~~Loop through the packets to be sent.~~
      2. ~~Set the TimeStamp Sync in every record just before sending.~~
      3. ~~Send the packet to the Server.~~
      4. ~~Wait for a confirmation (handshake) from the Sever that the packets have been processed.~~
         1. ~~The server will send fldID of the [\_Upload\_To\_Server] record processed, and the timestamp of processing (not sure for what the timestamp, but let’s send it for now).~~
      5. ~~With the fldID for each record confirmed as processed the system will:~~
         1. ~~Delete the record from [\_Upload\_To\_Server] table.~~

***~~Then, it seems that the TimeStamp Sync field can be eliminated altogether. Let’s see…~~***

1. ~~In this scheme, the Data Uplolad/Sync attribute becomes a~~ ***~~Table Attribute~~****~~that spills to Record Level~~* ~~because the uploads to Server must be done by record.~~

~~The TimeStamp Sync and the PushUpload fields in the Data tables become meaningless. They are defined and only meaningful in the [\_Upload\_To\_Server] table~~*~~. The Bitmask field needs to remain, though:~~*

~~Although the Bitmask can be queried from the table (Bitmask is an attribute for DataTables). For setRecord() Bitmask from the getTblName() function (~~*~~tbl\_bitmask~~* ~~is the 5~~~~th~~ ~~item in the return tuple): under certain circumstances, a field from an “uploadable” table might not be uploaded.~~ *~~This is case of records generated by system operations (Category, timeout, etc.)~~* ~~-> records generated by TimeInterval functions.~~

~~In these cases, the Bitmask bit will be set by the system function~~***~~: It is then essential that setRecord(), setRecords() retrieve the Upload/Sync attribute from the records themselves and not pull it directly from the Table Attribute.~~***

1. ~~With the operations INSERT, UPDATE, DELETE established for each data set in setRecord(), setRecords(), delRecord(), the actions for the Server are completely defined and there is no need for other actions.~~
2. ~~All the above applies to the sync’ing logic that shares data among devices without a Server connection.~~

~~Rule for sync’ing:~~

* 1. *~~Sync’ing among devices is done as long as the Upload to server is not completed~~*~~. The devices sync’d are stored in the record’s~~ *~~PushUpload field~~* ~~of the \_Upload\_To\_Server table.~~ **~~Once a record is uploaded to Server, the system halts all sync’ing attempts of that record,~~** ~~as the Server will now be in charge if pushing the record to the rest of devices remaining to be sync’d. This is to avoid convoluting unnecessarily the sync’ing logic.~~

**Chapter 3: Programmed Activities 🡺 Most of this chapter is deprecated. Jump to the end of Chapter** (Search for **05May23. Implemented Solution.) to access valid information.**

02Nov22:

Going into Programmed Activities.

- Must define signatures for Triggers that are stored in DB and prevent from creating duplicate triggers.

- Started the definition of triggers and Trigger signatures.

- Developed operate() function to run operations between 1, 2, 3 operands. All defined in Python.

03Nov22

Coming to terms on how to deal with comparisons to conditions:

- Comparison to conditions must be implemented at Class level for object behavior consistency and data integrity.

- As such, Comparison Operators can be stored in ACTIVITY TABLES (Animal Activity, Device Activity, Caravana Activity, etc). The "Operator" name is reserved for Columns holding Conditions Operators

so, entities other than Activities could eventually define their own comparison operators, with the condition is that they be defined at Class level (ie, same for all instances of a class)

- The OPERATOR column will be a JSON/List of the form {operator, } for each Activity in [Tabla Actividades Nombres] for the class. This allows for a generalized definition where an Object may

hold different comparison operators depending on the Activity.

- The Operator Dictionary is loaded in memory as a class attribute for the Object.

- Additional Operators: In addition to the DB-stored operators, additional comparison operators may be defined for code-generated Object attributes. They will be kept in the same Operators Dict.

Fixed the issue of int dictionary keys being converted to string during json encoding in dumps() function: a hook function is defined and passed to the json.loads() function

Hook Function used:

def json\_keys\_to\_int(x): # Como los int keys se pasan a str, hay que reconvertir a int al cargarlos.

# Convierte en int los dict keys convertibles a int. TENER EN CUENTA y NO USAR '24','7', etc. como keys.

if isinstance(x, dict):

aux\_dict = {}

for k in x:

try:

aux\_dict[int(k)] = x.get(k)

except ValueError:

aux\_dict[k] = x.get(k)

return aux\_dict

return x

Then: json.loads(json\_data.decode(), object\_hook=json\_keys\_to\_int) #

04Nov22

- Added Column TimeStamp Sync to ALL tables in the system, to manage the DB synchronization logic (along with the BitMask and PushUpload fields).

05Nov22

- Found a way to avert "database is locked" errors in test\_threading.py: run a loop trying to execute the PRAGMA command while db is busy, until it completes. See end of test\_threading.py. -> Update 25Apr23: Now all that’s managed by the destructors. The code has been removed.

By fixing the code in setRecord(), setRecords() functions, the ‘*database is locked’* error is averted.

Triggers:

Need some standardization to pass fields and data across objects for testing Trigger and PA conditions:

There are 3 sources of data fields:

1) Object itself (Animal, Person, etc). Data as dob, mf, animalClass, castrated, etc.

2) Activity Data (Programmed Activity, Trigger Activity).

3) Embedded dictionaries in [Data Actividad Programada] and in [Triggers] tables

To come to terms with this, some basic rules:

1) All data coming from objects must retain in the Trigger obj the attribute name in the original object.

2) All data coming from DB must conform to the standard fldUID definition.

And more rules to come...

\* All data from [Data Programacion De Actividades] to be loaded onto Trigger instances as a dictionary using fldUIDs for [Data Programacion...] fields.

06Nov22

More Trigger Comparison requirements:

- In general, it would be ok to ignore (remove from comparison) Trigger fields that are absent in Object trigger data (DOB, Age, mf).

- HOWEVER: There are the conditions that, if absent in an object, prevent the assignment of a Trigger to the object:

\* The definition must be based on Programmed Activity. For instance, AnimalHealth (Sanidad) activities cannot be applied if

the trigger has Localization defined and the Animal doesn't.

To define: data structures and logic to assign triggers based on Programmed Activities.

Quite some work done on iterating through nested dictionaries to pull data from special fields. Special fields are fields that don't compare with regular

operators (==, <, >, >=, etc).

For now, special field is Localization.

1) krnl\_abstract\_class\_activity.py: Iterator to go through nested dictionaries produced by DeepDiff and pull fields that require special comparison methods:

def nested\_dict\_iterator(dicto): # Que lindo, esta es un generator function (usa yield) y tambien funca...

for key, value in dicto.items():

if isinstance(value, dict): # Check if value is of dict type

yield {\*\*value} # yield the value dictionary and continue working...

nested\_dict\_iterator(value) # run function recursively and search for more values that are type dict.

else:

yield {key: value} # If value is not dict then yield key: value.

2) krnl\_abstract\_class\_activity.py: Implemented a dictionary of functions to execute special comparison methods:

i) \_\_specialFields = {'fldFK\_Localizacion': compareLocations}

ii) def compareLocations(loc1, loc2):

return Geo.compareLocations(loc2, loc1)

iii) Calling through the dictionary: once the special field is identified, the comparison method is called:

if ddKey:

values = d[ddKey].values()

result = self.\_\_specialFields[specialFldKey](\*values)

if result:

result = True if len(d) == 1 else False

3) Finally, the iterator used is one that "linearizes" or flattens all dictionaries to a list of tuples. This is the function:

def nested\_dict\_iterator\_gen2(dicto):

""" This function accepts a nested dictionary as argument and iterate over all values of nested dictionaries """

# Iterate over all key-value pairs of dict argument

for key, value in dicto.items():

if isinstance(value, dict): # Check if value is of dict type

for pair in nested\_dict\_iterator\_gen2(value): # If value is dict then iterate over all its values

yield (key, \*pair)

else:

yield (key, value) # If value is not dict type then yield the value

07Nov22:

- Worked on the trigger comparison functions, developing the functions for handling DeepDiff comparisons.

In order to compare, the dict of dicts returned by DeepDiff is "flattened" to a list by the iterator above. Then, there are 2 cases:

1) processMultiKeyNames(argDict): Function used to make comparision among fields with different field names. Those cases are listed in the dictionary

\_multiKeyNames = {'between': ('\_ageMonths', 'minAgeMonths', 'maxAgeMonths'), }

'between': Operator to be applied / ('\_ageMonths', 'minAgeMonths', 'maxAgeMonths'): Fields to pull from List for operator to be applied on.

The problem arises when traversing the list generated by the def nested\_dict\_iterator\_gen2() function, because the names (as defined in the dict above)

must be pulled, their associated values also pulled, and processing made with the operator defined for that set of key names.

processMultiKeyNames addressed this case.

2) Processing of special fields. These fields required special functions to be compared. They are defined in dict

\_specialFields = {'fldFK\_Localizacion': (compareLocations, 2), }

compareLocations is the function that processes the values; 2 is the # of arguments required by the function compareLocations.

Here, when the special field is found in the f

Pending: what to do with the rest of the fields returned by DeepDiff that don't fall in the 2 cases above. For now, they are all checked with ==.

But a logic will have to be developed for other particular fields as needs arise.

08Nov22:

**\*\* IMPORTANT: CHANGED DOB to Animals 1 and 11 (Females) in database GanadoSQLite 4.1.6 - Clean. Original values stored on adjacent field. Must change back when done with testing.**

More fixes to the Trigger comparison functions

- Added exception handling to the JSON conversion functions to avoid unhandled exceptions when the DB data doesn't conform to JSON

def adapt\_to\_json(data):

# TODO(cmt): OJO! Los dicts con int keys se pasan a str. Esto sucede con fldContainerTree de [Geo Entidades]

try:

return (json.dumps(data, cls=JSONEncoderSerializable, sort\_keys=True)).encode()

except(JSONDecodeError, Exception):

raise DBAccessError(f'JSON encoding error: {data}')

def convert\_json(json\_data):

try:

return json.loads(json\_data.decode(), object\_hook=json\_keys\_to\_int) #

except JSONDecodeError:

raise DBAccessError(f'JSON conversion error: {json\_data}')

- Generalization effort for operations on fields:

**\*\*\*\* RULES \*\*\*\***

\*\*\* All these rules are subject to change as needs arise \*\*\*

Given that there are few fields that require special treatment (processMultiKeyNames, Location field, etc) the design approch taken is:

1) Special fields are defined in dictionaries in krnl\_cfg\_items.py, FOR DeepDiff OBJECTS ONLY:

\_multiKeyNames {}

\_specialFields {} --> See if these 2 can be merged at a later stage, when the solution is better defined.

2) excludedFields= [] -> A list is defined for fields that require NO comparison. This is special a case enough to merit a dedicated data structure.

All fields that should not be compared in the general case are included in this list. AS A DESIGN criteria: if a field defined in this dictionary is also included in the \_specialFields dictionary, the definition in \_specialFields overrides the no-comparison criteria of skipFields.

Examples of non-comparison fields: 'fldComment', fld.

*As a general rule, all columns of type JSON should be excluded from direct comparison.*

3) \_specialFields = {fldName: (operator | function, # of\_args, contextDict={}), } Example: \_specialFields = {'fldFK\_Localizacion': (compareLocations, 2, {}), }

fldName: the field to be operated upon, coming from DB or created ad-hoc in the code.

operator: operator to apply to the field name

# of\_args: Number of args taken by the operator or function

contextDict: For extension purposes at this time. It would include context information that could change the behavior or override operator depending on context.

Note: 1, 3 defined on the prerogative that comparisons are done on objects returned by DeepDiff and a linearization (flattening) is done on the returned dictionaries using a dict\_iterator function.

4) All fields that do not fall in the 2 dictionaries and the excludedFields list described above, are compared via '==' operator (the default operator when some info is missing).

5) Strings: all strings are compared *case-insensitive*, converting to lower-case and stripping() before comparison. This includes characters (strings of len=1)

6) Numbers: all numbers are checked with the python-embedded logic, as it works. Example:

35 == 35.0000000000000000001 # decimal places > resolution

True

35 == 35.00000001 # decimal places < resolution

False

Resolution is set at 14 decimal places.

7) List, dict and set objects: are compared IGNORING the order of the elements. Each element is compared using the criteria defined for the type.

8) In comparisons of 'belonging' or 'is in' Operand1 (op1) is defined in all cases as the operand to check if *belongs to* or *is contained in* Operand2 (op2). So in the case of compareLocations(): operate(loc1, loc2) will check if loc1 is in/is contained in loc2.

\* Once better defined, implement a "Operation Data" column in \_sys\_Fields table to contain the operation data in JSON. If the column is left None/NULL -> it's a '==' field.

Also implement an "Excluded Field" column. 1->field is to be excluded from all comparisons (can overriden if the field is lated defined in \_specialFields dict).

More fixes to the function that compares DeepDiff dictionaries:

- Implemented \_specialFields, \_excludedFields, multiKeyNames dictionaries with their callback functions and methods.

09Nov22:

Rules start to change:

\*\*\*\* RULES iter #2 \*\*\*\*

\*\*\* All these rules are subject to change as needs arise \*\*\*

New approach: we're trying to scrap excludedFields, \_specialFields, etc.

1) Strings: all strings are compared case-insensitive, converting to lower-case and stripping() before comparison.

This includes characters (strings of len=1)

removeAccents(): in addition, all string operating as variables/arguments in the system are stripped from all accents, in a way that, for the system

"El Nandu" = "el ÑanDÚ" = "EL ñandü" = "el nandu". FOR ALL PURPOSES, the string used internally is "el nandu".

2) Numbers: all numbers are checked with the python-embedded logic, as it works. Example:

35 == 35.0000000000000000001 # decimal places > resolution

True

35 == 35.00000001 # decimal places < resolution

False

Resolution is set at 14 decimal places.

3) List, dict and set objects: are compared IGNORING the order of the elements. Each element is compared using the criteria defined for the type.

4) In comparisons of 'belonging' or 'is in' Operand1 (op1) is defined in all cases as the operand to check if belongs or is in Operand2 (op2). So in the

case of compareLocations(): operate(loc1, loc2) will check if loc1 is in/is contained in loc2.

5) Field Key Names:

All field key names used in the system will have 1 of 2 signatures:

1. DB Signature (fldUID): comprised of table name and field key name as follows: tblName\_\_fldKeyName. the joining char particle '\_\_' is programmable.

2. System Generated signature: fab#varName -> ad-hoc variables that need to be compared against DB key names are defined as fab# is a particle that identifies a valid system-generated particle when parsing through dictionaries and other data structures. Also programmable.

An immediate example of system-generated variable is age (for animals and other objects). The var name is fab#age.

***Follow up: with the current implementation of dict\_iterator\_gen(), it MAY NOT BE required to use the "fab#' particle in fieldNames.***

6) Comparison of data objects:

Objective:

The objective of the comparison of data objects in the system is to evaluate values corresponding to the SAME FIELDNAME, defined in the objects to compare.

Therefore, the parsing and comparison must be done at fieldName level, instead of dictionaries, lists, etc (A certain fieldName may be defined inside data structures of different type in each of the comparison objects, and still they must be compared if they share the same fieldName)

i) For the time being, only comparisons between 2 objects obj1 and obj2 is defined.

ii) Also, for the time being, it is assumed that only one value, the first value found is the valid value for a fieldName if that fieldName is repeated. This INCLUDES None values, as NULL coming from SQLite convert to None.

iii) Parsing: Parsing of the objects will be done confirming to the rules i, ii and data structures (dicts) will be generated for comparison.

The general form of these dicts will be {fldKeyName: fldValue, }

fldValue can be of any type, EXCEPT for dict (dicts are de-constructed to their {k: v} pairs to fill the comparison data structures.

iv) The data structures then will be compared after parsing both objects separately, applying the operators defined for each fldName and each comparison function.

7) Parsing: Not sure how or where DeepDiff will be used at this point.

For the rest of the parsing, the first approach to comparing is: the 2 dictionaries are searched for each keyName. 2 possibilities.

i) fldKeyName present in both comparison objects -> {fldKeyName: [valObj1, valObj2]} (operations on valObj1, valObj2 are applied at a later stage).

ii) fldKeyName in one structure but not in the other one (Unique field, for whatever reason) -> {fldKeyName: val}

\* In the case the same fldKeyName appears repeated in the datastructure with different values, the value of the FIRST instance will be taken as valid.

The rest of the values from other instances of that fldKeyName will be ignored. This is because the objective is to compare fldKeyNames and values in DIFFERENT datastructures (from obj1 and obj2). It's like this for now, and it may change in the future.

To implement this, a generator function that splits dictionaries in its component (nested) dictionaries and returns a list of dicts will be used.

The idea is to traverse each of the dictionaries in the lists and generate results dictionaries with the form {fldKeyName: [valObj1, valObj2]} for each fldKeyName found to compare, following the criteria defined in 6.

8) \_multiKeyNames {} - defined in krnl\_cfg\_items.py. Dictionary of the form \_multiKeyNames = {'between': ('\_ageDays', 'minAgeMonths', 'maxAgeMonths'), }

The structure defines and operator ('between') and fields that are related, in the order listed by that operator.

*By all means try to avoid its use, going to simpler handling, but the feature must be made available in case of need.*

9) Defined for fields that require special comparison conditions:

\_specialFields = {fldName: (operator | function, #of\_args, contextDict={}), }. Ex: \_specialFields = {'fldFK\_Localizacion': (compareLocations, 2, {}), }

fldName: the field to be operated upon, coming from DB or created ad-hoc in the code.

operator: operator to apply to the field name

#\_of\_args: Number of args taken by the operator or function

contextDict: For extension purposes at this time. It would include context information that could change the behavior or override operator depending

on context.

10) excludedFields= [] -> A list is defined for fields that require NO comparison. This is special a case enough to merit a dedicated data structure.

All fields that should not be compared in the general case are included in this list. AS A DESIGN criteria: if a field defined in this dictionary is also included in the \_specialFields dictionary, the definition in \_specialFields overrides the no-comparison criteria of skipFields.

Examples of non-comparison fields: 'fldComment', fldBitmask, fldTimeStampSync, fldPushUpload, etc.

*As a general rule, all columns of type JSON should be excluded from direct comparison.*

11) Every object processing DB Data should implement a method getCompData() -> dict. Returning a dict of all the "comparable" fields for the object.

The dict should be of the form {fldName: fldValue, } for all the fldNames to be included in the comparison of such object.

The dict returned by getCompData() is fed to the comparison function.

12***) Field Name Equivalence Mapping***: Dictionary with all possible field names equivalence to assign fieldNames to operations. Example:

\_fldEquivalence = {'\_lastInventory': ('tblDataAnimalesActividadInventario\_\_fldDate', ),

'\_lastStatus': ('fldStatus', 'fldFK\_Status'),

'\_lastLocalization': ('fldFK\_Localizacion', 'fldFKLocalizacionOrigen', 'fldFKLocalizacionDestino'), }

With this, all fields are searched in the equivalence dictionary to assign to the correct operation, matched with other possible field names for the same variable.

\* TODO: IMPLEMENT RANGES, so that age comparisons, time-window verifications can be done using 2 parameters with the same name, and the \_multiKeyNames logic is avoided.

Example: fab#age is age. fab#range = (maxAge-minAge). In this case, the custom function withinRange() can be used instead of using multiKeyNames.

==> withinRange(fab#age, fab#range)

11Nov22:

New implementation for functions, based on the rules (iter #2):

- The design criteria is to use compareXXX() functions for all XXX objects that require DB fields comparisons: compareTriggers(), compareActivities(), etc.

These compare functions will encapsulate all the details and nuances for performing valid object comparisons.

- This is mainly aimed, for the time being, to be used for Programmed Activitites. The idea is that the concept could be extended to other areas if needed.

- The compareXXX() functions return a dictionary of the form {fldName: comparison\_result (True/False), }.

The idea is that the calling function can iterate over each field name and take actions on the individual results of the comparisons at fldName level.

By default, the list of values of the return dictionary turn into True only if all elements of the values array are True. But again, the idea is that this can be modified ad-hoc if need be.

\* TODO: DESIGN CRITERIA: Once the DataBase structure is set, switch over to all lowercase naming, to simplify string comparison.

12Nov22:

More changes to the rules, and a more consistent version of compareTriggers. Updates and corrections made to several functions to support compareTriggers.

Rules changes:

Rule #4 changes to the inverted assignment of operators as follows:

4) In comparisons of 'belonging' or 'is in' Operand1 (op1) is defined in all cases as the "containing" operand. The verification done is then if Operand 1 CONTAINS Operand 2 (op2) case of compareLocations(): operate(loc1, loc2) will check if loc1 contains (or is equal to) loc2.

8) Rule #8 is removed. No more multiKeyNames, it's no longer necessary.

With this, the new rules are:

\*\*\*\* RULES iter #3 \*\*\*\*

\*\*\* All these rules are subject to change as needs arise \*\*\*

multiKeyNames removed.

1) Strings: all strings are compared case-insensitive, converting to lower-case and stripping() before comparison.

This includes characters (strings of len=1)

removeAccents(): in addition, all string operating as variables/arguments in the system are stripped from all accents, in a way that, for the system

"El Nandu" = "el ÑanDÚ" = "EL ñandü" = "el nandu". FOR ALL PURPOSES, the string used internally is "el nandu".

2) Numbers: all numbers are checked with the python-embedded logic, as it works. Example:

35 == 35.0000000000000000001 # decimal places > resolution

True

35 == 35.00000001 # decimal places < resolution

False

Resolution is set at 14 decimal places.

3) List, dict and set objects: are compared IGNORING the order of the elements. Each element is compared using the criteria defined for the type.

4) In comparisons of 'belonging' or 'is in' Operand1 is defined in all cases as the "containing" operand. The verification done is then if Operand 1 (op1)

CONTAINS Operand 2 (op2). In the case of compareLocations(): operate(loc1, loc2) will check if loc1 contains (or is equal to) loc2.

5) Field Key Names:

All field key names used in the sistem will have 1 of 2 signatures:

1. DB Signature (fldUID): comprised of table name and field key name as follows: tblName\_\_fldKeyName. the junction char particle '\_\_' is programmable.

2. System Generated signature: fab#varName -> ad-hoc variables that need to be compared against DB key names are defined as

fab# is a particle that identifies a valid system-generated particle when parsing through dictionaries and other data structures. Also programmable.

An immediate example of system-generated variable is age (for animals and other objects). The var name is fab#age.

Follow up: with the current implementation of dict\_iterator\_gen(), it MAY NOT BE required to use the "fab#' particle in fieldNames.

6) Comparison of data objects:

Objective:

The objective of the comparison of data objects in the system is to evaluate values corresponding to the SAME FIELDNAME, defined in the objects to compare.

Then, the parsing and comparison must be done at fieldName level, instead of dictionaries, lists, etc (A certain fieldName may be defined inside

data structures of different type in each of the comparison objects, and still they must be compared if they share the same fieldName)

i) For the time being, only comparisons between 2 objects obj1 and obj is defined.

ii) Also, for the time being, it is assumed that only one value, the first value foun is the valid value for a fieldName if that

fieldName is repeated. This INCLUDES None values, as NULL coming from SQLite convert to None.

iii) Parsing: Parsing of the objects will be done confirming to the rules i, ii and data structures (dicts) will be generated for comparison.

The general form of these dicts will be {fldKeyName: fldValue, }

fldValue can be of any type, EXCEPT for dict (dicts and de-constructed to their {k: v} pairs to fill the comparison data structures.

iv) The data structures then will be compared after parsing both objects separately, applying the operators defined for each fldName and each

comparison function.

7) Parsing: Not sure how or where DeepDiff will be used at this point.

For the rest of the parsing, the first approach to comparing is: the 2 dictionaries are searched for each keyName. 2 possibilities.

i) fldKeyName present in both comparison objects -> {fldKeyName: [valObj1, valObj2]} (operations on valObj1, valObj2 are applied at a later stage).

ii) fldKeyName in one structure but not in the other one (Unique field, for whatever reason) -> {fldKeyName: val}

\* In the case the same fldKeyName appears repeated in the datastructure with different values, the value of the FIRST instance will be taken as valid.

The rest of the values from other instances of that fldKeyName will be ignored. This is because the objective is to compare fldKeyNames and values

in DIFFERENT datastructures (from obj1 and obj2). It's like this for now, and it may change in the future.

To implement this, a generator function that splits dictionaries in its component (nested) dictionaries and returns a list of dicts will be used.

The idea is to traverse each of the dictionaries in the lists and generate results dictionaries with the form {fldKeyName: [valObj1, valObj2]} for each

fldKeyName found to compare, following the criteria defined in 6.

8) Defined for fields that require special comparsion conditions:

\_specialFields = {fldName: (operator | function, #of\_args, contextDict={}), }. Ex: \_specialFields = {'fldFK\_Localizacion': (compareLocations, 2, {}), }

fldName: the field to be operated upon, coming from DB or created ad-hoc in the code.

operator: operator to apply to the field name

#\_of\_args: Number of args taken by the operator or function

contextDict: For extension purposes at this time. It would include context information that could change the behavior or override operator depending

on context.

9) excludedFields= [] -> A list is defined for fields that require NO comparison. This is special a case enough to merit a dedicated data structure.

All fields that should not be compared in the general case are included in this list. AS A DESIGN criteria: if a field defined in this dictionary is also included in the \_specialFields dictionary, the definition in \_specialFields overrides the no-comparison criteria of skipFields.

Examples of non-comparison fields: 'fldComment', fldBitmask, fldTimeStampSync, fldPushUpload, etc.

As a general rule, all columns of type JSON should be excluded from direct comparison.

10) Every object processing DB Data should implement a method getCompData() -> dict. Returning a dict of all the "comparable" fields for the object.

The dict should be of the form {fldName: fldValue, } for all the fldNames to be included in the comparison of such object.

The dict returned by getCompData() is fed to the comparison function.

11) Field Name Equivalence: Dictionary with all possible field names equivalence to assign fieldNames to operations. Example:

\_fldEquivalence = {'\_lastInventory': ('tblDataAnimalesActividadInventario\_\_fldDate', ),

'\_lastStatus': ('fldStatus', 'fldFK\_Status'),

'\_lastLocalization': ('fldFK\_Localizacion', 'fldFKLocalizacionOrigen', 'fldFKLocalizacionDestino'), }

With this, all fields are searched in the equivalence dictionary to assign to the correct operation, matched with other possible field names for the same

variable.

12) Exact Match in Trigger Comparisons: In the case of triggers, 2 comparisons are defined:

- Exact Match (exact\_match=True): comparison between 2 trigger objects (executed when the obj parameter is not None): Compares all fields of both triggers,

irrespective of their values.

- Loose Match (exact\_match=False): used when comparing a trigger (pointed to by self) with a dictionary. In this case, all variables with value None in the trigger are ignored.

The logic implemented with this match type is that when a condition must be ignored in a trigger definition, it's value be set to None. Then, when comparing

trigger vs trigger (where all fields matter) the "None" fields are compared anyway. But when a comparison is done with an object, the None fields are ignored because those particular fields are not to be counted when determining if the trigger will be assigned (or not) to the object.

13Nov22:

Meeting Trigger conditions:

- Object INSTANCES (Animal, Person, etc) are the entities meeting trigger conditions: the checks must be made for each of these instances. The conditions to be met are defined in the instance attribute. The function is ActivityTrigger attribute: triggerConditionsMet(self)

- Once conditions are met, a Programmed Activity (PA) has to be created for the object instances with the parameters defined within the trigger.

How all this pans out with objects, triggers and activities:

1) Trigger Creation:

Triggers are created and stored in triggerRegisterDict for each applicable object class (Triggers for Animals, for Person, for Device, etc). When a trigger is

created and registered:

- A loop is run for all active objects in the objectRegisterDict and the trigger is checked against each of the object's conditions:

- When all conditions are met, the new trigger is assigned to the object that meets the conditions.

- This same routine and code is executed during initialization to assign database-stored triggers to the respective objects.

2) Object Creation:

When a new object is created and registered in the objectRegisterDict.

- A loop is run through all the active triggers in the triggerRegisterDict and all triggers whose conditions are met by the objects are assigned to the object.

3) Background Stuff:

There are a lot of conditions that involve every trigger and activity (and expected to grow over time), so to check for condition changes individually is unfeasible.

Also, several critical conditions change their state with the passage of time (Category, age, status, etc).

Then, the checks for trigger (and Activity) condition fulfillment will be performed by dedicated background functions, running in one of the several timer threads:

- TRIGER MATCHING: One function will loop through ALL objects (example: Animals) and all Triggers and will do conditions-matching and will assign triggers to objects that meet the trigger conditions.

- PA CREATION: Right after that, and in the same loop, another function will do a similar check between objects and PA Conditions: when conditions are met to

create a PA, the function will create a PA for the object and will insert:

a) 1 record in table RAP with the definition of the Programmed Activity being created. This record is created ONLY for the 1st object (1st Animal, etc) for which the Activity is defined. For the rest of the objects that are candidates for the same Programmed Activity there will be entries in tblLinkPA, as described below in b).

b) 1 record in tblLinkPA with a link to the Activity just created and the object ID (fldFK field in the tblLinkPA) to assign to the object the Activity just created.

Function name: paCreateActivity().

4) PA Closure:

This check will be performed with every function that executes an activity and records it in DB (usually set() functions that there can be other ones).

When an activity is performed/executed a function will be called to cross-check this activity against the PA list for the object for which the activity is performed:

- This function will loop through all the PA records from DB defined for the object that executes the activity and if an activity is found to conform to conditions. 25Apr23: Loop only through PA records that DON’T have status=Retired.

Status=Retired flags PA records that are long overdue and are not loaded in memory during startup. They remain in DB only for record tracking purposes.

the PA will be closed with the data of the performed activity as Closing Activity.

- Function paMatchAndClose(). It should take no arguments and return nothing.

To close an Activity for 1 or n objects:

a) the match function checks on the conditions of the activity.

b) if ALL conditions match, then the match function searches the tblLinkPA and searches all objects that have that particular PA defined, and sets closure

on each of those records in the tblLinkPA, writing an UPDATE operation on field "ID\_Actividad De Cierre". All other fields remain unchanged.

\*TODO: DEFINE LOGIC FOR EQUIVALENT FIELDS: A logic to create standard equivalent fields starting from a field name that may or may not be a fldUID.

15Nov22:

- Moved field Sequence (Secuencia) from table Triggers to table [Programacion de Actividades] (Sequence is a property of the Activity itself, not the Trigger).

- For now, PA attributes are created in Activity. \_\_init\_\_ () when the supportsPA attribute is set to True. Let's see how this pans out.

Programmed Activities. More generalizations:

- IMPORTANT: \_activityData Dictionary: defined in Activity class.

A dictionary with ALL the data passed to execute the activity. Must take some kind of standard form.

All the data in this dict must map to Programmed Activities, so that the parameters of PA and the parameters from the executed activity can be matched.

As the data handled by 1 activity may involve multiple DB tables, all the keynames in the \_activityData dict must be of the form fldUID.

- Define an argument "supports\_pa" for all activity functions. If the argument is set to True, the pa routines are executed for the method. The method chooses when. -> DONE

- Define a dictionary for ALL Activity Singletons of the form {param\_name: fldUID}

This dict will be used by the paMethods() to assign values and to run comparisons/matches against programmed activities. It's a way to automate argument-setting, and argument-matching between executed activities and programmed activities.

16Nov22

Check for Execution Date of a PA: To verify that one activity executed on a certain date qualifies as the closing activity for a PA, the following logic must be implemented, based on the currently defined parameters:

tblLinkAnimalesActividadesProgramadas\_\_fldBaseDate: Actual Execution Date (when activity was performed)

tblDataProgramacionDeActividades\_\_ fldDate: Date the PA record was created (Reference for PA).

tblDataProgramacionDeActividades\_\_ fldDiasParaEjecucion: time span in days, counting from \_\_fldDate, when the activity is expected to be executed.

tblDataProgramacionDeActividades\_\_ fldWindowLowerLimit: Lower Limit counting from \_\_ fldDate to execute the activity.

tblDataProgramacionDeActividades\_\_ fldWindowUpperLimit: Upper Limit counting from \_\_fldDate to execute the activity.

fldWindowUpperLimit fldProgrammedDate executedDate ldWindowUpperLimit

(tblDataProgramacion) (tblLInkPA) (tblAnyActivity) (tblDataProgramacion)

18Nov22

~~Tables Actividades Nombres: Changed the field “Signature YN” to “Supports PA” to identify all Activities that support Programmed Activities.~~

~~supportsPA now is defined in the Actividades Nombres tables and imported directly into the activity objects. Initialized in \_\_init\_\_().~~

~~Main functions for triggers and PA fully defined (1~~~~st~~ ~~pass).~~

~~Now need to start testing some PA.~~

~~25Apr23~~

~~Define dictionaries~~ *~~criteria~~* ~~and~~ *~~conditions~~*~~. These will be stored in table~~ *~~[Data Programacion De Actividades]~~* ~~as 2 JSON fields.~~

~~Criteria Dictionary: Criteria are definitions that are~~ ***~~external~~*** ~~to the object (Animal, Device, Tag, etc).~~

~~- Geography (Localization)~~

~~- Time Range.~~

~~Conditions Dictionary: Conditions are definitions and features that are~~ ***~~attributes of the object.~~***

* ~~Age -> Range~~
* ~~Gender -> Male, Female, All~~
* ~~Race: Single value or list of Races.~~
* ~~animalKind: Single value~~
* ~~AnimalID -> Single value or List of IDs.~~

*~~Is this really needed, or using “Data Actividad Programada” (JSON field) will be enough? NOT CLEAR FOR NOW.~~*

**~~Argument Passing~~**~~: In order to accommodate for varying number of arguments in operations, implement the form:~~

~~[operand, argument\_tuple]~~

~~where argument\_tuple is (arg1, arg2, arg3, ).~~

26Apr23: EQUIVALENT FIELD NAMES

**Situation**: A certain parameter may take different names in the system, as it will take different field names in the DB (fldStatus, fldFK\_Status, 'fldFK\_Localizacion', 'fldFKLocalizacionOrigen', 'fldFKLocalizacionDestino', etc) and the system may use separate, different names to refer internally to those parameters or variables.

The objective of the devised logic is to identify and normalize all the different names referring to the same variable or argument and assign a unique name for it so that its processing is simplified in the code.

**Breakthrough idea:** Define ***Operand class*** with all attributes to operate over all fields in the system (DB fields and non-DB field names).

All Operands compare with regular operators, just by their operand name. The user must know what is being compared.

For very particular operands, overloaded or new operations are defined. They should be kept to a minimum.

**An even more breakthrough idea: don’t use Operands at all.**

All checks needed are between int, float, datetime and str mainly, with standard operators.

This could be done using just 2 lists: linkedNames and equivalentFieldNames. *See how this can be implemented and whether Operands could effectively be avoided.*

The problem boils down to finding the right fields from 2 dictionaries to compare/check and the operator to ‘operate’ them.

Default operator is == for all fields.

If a different operator is to be used among 2 or more fields of the 2 activities being compared, ***those operators must be defined in the Activity object itself*** because each Activity determines how its parameters must be compared. *Same goes for Triggers.*

“Compare functions” to be defined are:

* Activity.compareActivities()
* Trigger.compareTriggers()
* Geo.compareLocations()
* Activity.matchClosingActivity()

Function matchClosingActivity(closingActivity) *is not* commutative by nature. So no need to try to make operators and operations commutative. Order of activities matter.

Compare Functions should will compare all fields to equal except for:

* Excluded Fields.
* Fields for which specific operators are passed to the function -> See how to implement this.

**Another approach. Just an idea, not the final implementation.**

The general situation is that selected fields (their names and their values) coming from 2 objects (Activities, Triggers, etc) must be compared.

When the processing is checking for equal, the solution is trivial.

When the processing requires for special functions:

* The function to use for processing is defined by dictionary \_specialFields, which has the structure shown below
* Field names (Field Equivalent Name) are associated with the processing function, via dictionary specialFields{}. Each specialFields entry contains the names of the required arguments in the structure shown below:

*\_specialFields={'fldName': (opr | function, (key\_names\_op1,), (key\_names\_op1,), {contextDict})*

args\_names\_op1, args\_names\_op2: key names to be picked from operand1 and operand2 to pass to the processing function. Values are obtained from the objects/dictionaries passed as argument to the main function (isClosingActivity() or compareTriggers())

Any required field not present is passed as None and the processing function must handle it.

2 Arguments passed None value or False value compare as True.

Non-valid arguments (keys missing in dictionaries) are skipped. They do NOT return False, as such comparison is invalid.

TODO**:** Define background function to retire old/obsolete programmed activities that are non-permantent.

* Run the function daily or twice a day and set DB Field for progActivity as “Retired”
* Remove the activity from the Prog. Activities Register in memory so that the retired activity is not checked for anymore.

**05May23. Implemented Solution.**

A class ProgActivity() is defined. All programmed activities Classes are derived from it: ProgActivityAnimal(), ProgActivityTag(), etc.

This class ProgActivity() HAS NO RELATION with Class Activity(): THIS IS THE NEW CONCEPT.

The ProgActivity() class encapsulates the interface with the ActivityTrigger() class.

All triggers are handled by ProgActivity() methods. Triggers should not be handled by the Activity() class.

The idea is that executed/performed activities are managed by the respective activities singletons.

On the other hand, ProgActivities is a dictionary with all programmed activities listed. The ProgActivity() methods handle the interface with DB tables and the functions to compare ProgActivities.

The Activity() class implements the method isClosingActivity() to determine whether a given performed activity is a closing activity for one of the ProgActivity objects in the ProgActivity() dictionary of programmed activities.

Triggers: the object IDs (AnimalID, TagID, etc) should **not** be a comparison element to determine belonging or equality. In general, a trigger object must have and handle a list of objects (Animals, etc) that will in turn generate Programmed Activities when the trigger conditions are met.

Thus, the objects should only check and compare their assigned programmed activities and not relate in any way to triggers, which are a stage prior to a programmed activities.

All triggers should be managed by the ProgActivity() class methods.

**Important concept**: When a ProgActivity object is created (via direct creation or via triggers), the ‘target population’ of objects for which it is created is defined there and then.

Later in time, objects may be added or removed from that list due to changes in object state or in the activity state itself (completion, for instance).

Then, an activity should store in a variable the target population it applies to.

***The problem of up-to-date progActivities.***

A method must be devised to keep track of the target population of all the active Programmed Activities, or rather, for each object (Animal, etc) to keep a list of the progActivities assigned to it.

*The first approach is to attach Prog. Activities to Objects (Animal, Person, etc) as the object is the one “requiring” or assigned the Programmed Activity.*

Because of the way the objects enter and exit the system, it is simpler and more efficient to attach defined ProgActivities to each object in the system.

* A list (set) will be kept at class level with the list of all “active” ProgActivities. An active ProgActivity is an activity that is listed in at least 1 of the objects present in the Object registerDict.
* Another list will be kept for each object (Animal, etc) with the progActivities active for that particular object.

The addition of all the individual object lists will feed the overall class progActivities List (It ***must*** be a list because each entry will be eventually matched with a removal when the progActivity is closed. Then, if a progActivity is registered more than once, then it must be closed by different objects and activities more than once too).

* ProgActivities can have 3 states (defined in the Activity Flag):

0: Inactive. Not loaded in the system.

1: Active.

2: Permanent (the activity must be always loaded and checked against to apply to all pertinent objects).

* When an Active (state=1) Activity no longer appears in the activeActivitiesRegister, it can be removed from the ActivitiesRegister and its status set to Inactive. The ProgActivity will remain in the DB but will no longer be loaded in the system to be checked against performed activities.

Objects (Animal, Tag, Person, etc) must have 2 lists associated to each instance:

* ProgActivites: A list of all the already programmed activities for the object.
* triggerActivities: A list of all the activities qualify as a Trigger that will create in turn a programmed activity for the object, to be executed when the conditions are met.

These are implemented via a dedicated “dirty” table in DB [Animales Listas De Programacion] that stores the 2 lists above for each Animal object (in this example).

The data in these tables is translated to the system to 1 instance variable (\_\_myProgActivities) and 1 class variable (\_\_activeProgActivities).

**24-May-23:**

**progActivity Objects ID:** ProgActivity objects are created by reading data from DB or by the system.

In order to create progActivity objects, loadFromDB(cls) must be invoked with the right ProgActivityClass (ProgActivityAnimal, ProgActivityPerson, etc).

**ACTIVITIES:**

Creating a Generic Activity Class to be able to quickly define activities (Destete, Medicion, Movimiento, etc, etc).

With this class, activity objects are created in Animal class, when defining the property objects for the activities.

**25-May-2023: The problems with dynamically assigned attributes (\_outerAttr stack) in a multi-threaded environment.**

2 problems arise with the use of the \_outerAttr stack and the Activity.dynamic\_attr\_wrapper decorator that enables Activity Class code re-entry:

1. External call (via @property) of a class method that is NOT decorated. Example:

status.myFunc(), where myFunc() is not decorated with Activity.dynamic\_attr\_wrapper. In this case, the \_outerAttr stack will be incremented on entry but not decremented on exit of the call (since the dynamic\_atrr\_wrapper is not run).

**Solution:** Make all the non-decorated functions/method INTERNAL via \_\_funcName

In this way, only decorated methods can be accessed by objects from outside the Activity Class. -> This will be very contorted to implement due to access to attributes in parent classes (think properties like tblRAName, tblLinkName, etc defined in parent Abstract Activity Class). An alternative must be found.

1. The mirror problem: A method decorated with Activity.dynamic\_attr\_wrapper is called from *within* the Activity class via ***self***.

In this case, the \_outerAttr stack is not incremented (since the code to increment \_outerAttr is executed in the @property definitions) but the stack is decremented when the dynamic\_attr\_wrapper is executed.

**Solution:** define an attribute activity\_calls\_counter in class Activity. Logic:

This counter increments in dynamic\_attr\_wrapper BEFORE the method call. ***It is incremented both by object calls and by self calls.***

The condition is that when the size of the stack / len(self.\_outerAttr) is equal to self.activity\_calls\_counter, then the function was called from an object and the last element in the \_outerAttr stack is popped.

If the method is called via self, outerObject is not invoked hence activity\_calls\_counter *is* incremented but \_outerAttr stack is not (no objects added to \_outerAttr).

In this case, after the execution of the call, len(self.\_outerAttr) and activity\_calls\_counter will differ, so no items are popped from \_outerAttr stack.

The code from dynamic\_attr\_wrapper is shown below.

if self.\_outerAttr and len(self.\_outerAttr) == self.activity\_calls\_counter:  
 with self.\_\_lock:  
 self.\_outerAttr.pop() # Removes the last object from \_\_outerAttr list in Activity object.  
 self.activity\_calls\_counter -= 1  
else:  
 with self.\_\_lock:  
 self.activity\_calls\_counter -= 1

The constraint with this model remains that all calls made from objects (that invoke the outerObject property) MUST be done on functions decorated with dynamic\_attr\_wrapper, otherwise the logic driven by activity\_calls\_counter == len(self.\_outerAttr) will not work. -> Use a \_\_stack\_diff attribute set in the call to outerObject ???

The resulting code is:

def wrapper(self, \*args, \*\*kwargs):  
 with self.\_\_lock:  
 self.\_\_activity\_calls\_counter += 1 # This increment is not an atomic operation, hence the lock.  
  
 a = func(self, \*args, \*\*kwargs)  
  
 if self.\_outerAttr:  
 with self.\_\_lock:  
 if len(self.\_outerAttr) - self.\_\_activity\_calls\_counter == self.\_\_stack\_diff:

# Removes the last object from \_\_outerAttr list in Activity object.  
 self.\_outerAttr.pop()   
 self.\_\_activity\_calls\_counter -= 1  
 else:  
 with self.\_\_lock:  
 self.\_\_activity\_calls\_counter -= 1  
 return a  
return wrapper

So far, seems to be working…

But still the issue remains of an object call that invokes outerObject() made on a member function that is NOT decorated with *dynamic\_attr\_wrapper*: If the call is done without any nesting/re-entry, a ‘loose’ item will be left in the stack that will not affect system integrity. Any items left in the stack like that will be handled by the code above.

However, if the call is done in the middle of another call that uses the \_outerAttr stack (simultaneous access of the code), a problem emerges: if the 2nd call leaves an item in the \_outerAttr stack, the 1st call will fetch the wrong outerObject value from the stack when it resumes execution.

This can be only fixed by limiting the access of object instances ONLY to Activity Class members that are decorated with *dynamic\_attr\_wrapper.*

**21-Jun-23: All of the above IS DEPRECATED and was replaced with a \_\_outerAttr dictionary {current\_thread\_id: outer\_object, } that inserts one key for each thread calling the Activity methods.**

**The outerObject @property returns the outer\_object associated to the executing thread. This resolves all the problems associated with thread switching in the middle of execution of any Activity method.**

**30-May-23: SOME RULES FOR Fields Comparisons.**

Implementing the general concepts of **Deviation** and **Equivalences**

**Deviation:** A numeric value expressed as a *positive amount of value units* used to compare a delta or deviation from a center value.

Value – Deviation <= checkVal <= Value + Deviation

***Criteria:***

Deviation = None -> IGNORE. No checks are performed on Deviation

Deviation = 0 -> Exact Match of Value and checkVal

Deviation != 0 -> Deviation allowed as defined above.

*Deviation always positive and always expressed as absolute units of Value and checkVal, so that the operation Value – Deviation <= checkVal <= Value + Deviation always holds.*

**Equivalences**: A list of objects that admit the “in” operator.

True if checkVal in Equivalences / False if checkVal not in Equivalences

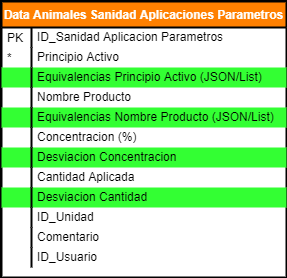
***Criteria:***

Equivalences = None or Empty List/Tuple () -> IGNORE. No checks made on Equivalences

Equivalences = () Empty List/Tuple - > EXACT MATCH (No Equivalences allowed). The same as setting Equivalences = (checkVal, )

Equivalences ! = () -> check with **in** operator.

Example:



* *Nombre Producto, Principio Activo* will have a list of strings defined in Equivalences to compare as True (belonging to).
* *Concentracion, Cantidad Aplicada* will have a number (%) allowed to compare as True.

With this general concept, we aim to narrow and define simple rules for the comparisons to be performed among fields of executed Activities and progActivities.

**5-Jun-23:**

**Main Comparison Criterion for progActivities (execute Data vs. proactivity Data):**

**executeFields will determine which fields to compare with the 3 rules below:**

**1. If a field is present in executeFields, the comparison is done with paFields.**

**2. If the field in executeFields is not present in paFields the comparison is False.**

**3. Fields not present in executeFields are ignored in all comparisons.**

*The concept is that paFields are a general template of comparison data related to a certain activity. They apply to a number of objects (Animal, Person, etc.).*

executeFields, on the other hand, carry the specific information of an Activity performed for a specific object, at a certain point in time: all the relevant data to make the comparison is better known at the time of executing the activity.

**11Jun23:**

**ProActivities Programmed, Execution Dates Definitions.**

Closing a progActivity involves 4 steps:

* UPDATE the tblLinkPA record (fldActividadDeCierre) with the id of the closing activity.
* INSERT record in[ tblDataXXXXActividadesProgramadasStatus] with the closing status data.
* Remove the progActivity Object from the executor’s myProgActivities dict.
* Run checkAndClose() method to determine is the progActivity record in tblRAP must be set to inactive (isActive=0).

Closing Status logic.

* A dedicated function, getPAClosingStatus() returns the status in which a progActivity is closed. If it returns None, the activity is ***not*** to be closed.
* Expiration Date is the absolute upper limit in which a progActivity qualifies to be closed by an executed activity, if the execution date is below or equal the Expiration Date.

If an executed activity complies with all the conditions required but its execution date is beyond the progActivity Expiration Date, the progActivity is not closed.

The closing rules are:

* + Executed between *lowerWindow* and *upperWindow*: Closed in time.
  + Executed between *upperWindow* and *ExpirationDate*: Closed Expired.
  + Executed after *ExpirationDate*: nothing’s closed.
  + A background thread scans the progActivity structure in the system and closes automatically (with closingStatus = ‘closedNotExecuted’) all open progActivities that have not been closed within a timeframe defined by the progActivity *timeout* attribute (isActiveFlag=1 because isActiveFlag=2 signals permanent activities that are not to be closed

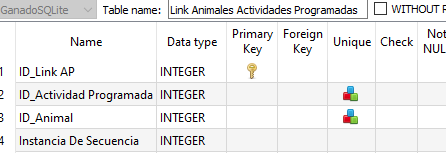
lowerWindow Programmed Date upperWindow ExpirationDate

*closedInTime closedExpired*

**Sequences.**

*A Sequence is a list of programmed activities to be executed in a predefined order, which is established by dedicated parameter (Ordinal field in the Sequence table).*

* All sequences are identified and accessed via a *unique name*. If not provided, the system must identify the sequence by assigning it a name.
* A sequence can be user-created or system-generated. In both cases a Sequence record is inserted the [Animales AP Secuencias] table, and all the progActivities tied to the sequence are created and inserted as records in the [Animales AP Secuencias Actividades - TEMPLATE] table.
* Sequences may (or may not) be assigned a Treatment record. Treatments are a special type of sequence that define sanitary/medical parameters for the activities.
* A function creates a list of progActivities in the [Animales Registro De Actividades Programadas] table and sets all the parameters for each progActivity using the template data from [Animales AP Secuencias Actividades – TEMPLATE] table.
* Since the records in [Link Animales Actividades Programadas] have Unique constraints as shown below, 2 records with the same ID\_Actividad Programada and ID\_Animal values are not allowed in the database.



Hence, sequence “instances” must be generated.

This is to be implemented by checking the existence of a ID\_Actividad Programada/ ID\_Animal pair present in table [Link Animales Actividades Programadas]. If the pair already exists (run a getRecords() search with the pair as parameters), the generating function will generate, *using the same Sequence record*, a new set of progActivities and insert them into the [Animales Registro De Actividades Programadas] table to be assigned to the ID\_Animal object that already had such Sequence assigned.

This will operate in fact as the “second instance” of the Sequence in question.

At the time, the need to define an actual Sequence Instance parameter/db field is not clear. If required, it must be defined in the [Link Animales Actividades Programadas] table.

**17Jun23**:

Tratamientos Tables provide is a set of sanitary descriptors that help define a Sequence.

A Sequence may or may not have an associated Treatment. If assigned a Treatment, the Treatment tables are used to pull information to define the Sequence ProgActivities.

**Recurrence. 29Jun23**

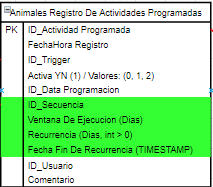
*A property of Programmed Activities* that prescribes that the Activity must be ***repeatedly*** programmed (and then executed) every certain number of days. Recurrence periods are established in days (unless otherwise specifically indicated).

Since recurrence is a property of the ProgActivity, if recurrence is indicated for a Sequence, the recurrence period is transferred/applied to each of the Activities of the Sequence.

2 database parameters manage recurrence:

* **Recurrencia (Dias):** Integer > 0 that establishes the number of days to elapse between two instances of execution of the same Activity.
* **Fecha Fin De Recurrencia:** datetime. A limit date. When programming the execution of the next instance of the Activity, if the ProgrammedDate is later than *Fecha Fin De Recurrencia*, then the established time for execution of the Activity has elapsed and the ProgActivity must be closed, by setting fldFlag = 0.

The parameters are defined in table [*Animales Registro De Actividades Programadas*]



Recurrencia:

Default value: 0. -> Indicates that the Activity is a one-timer. Closed once executed.

Fecha Fin De Reurrencia

Until when the ProgActivity Will continue to be programmed.

Defalut value: NULL -> If Recurrencia == 0: Ignore. Do nothing.

-> If Recurrencia > 0: Continuous Activity (same as fldFlag=2).

!=NULL**:** Date when programming recurrent Activities must stop.

If ProgActivity is part of a Sequence, *Fecha Fin De Recurrencia* must take into account the ProgrammedDate of the LAST Activity of the sequence. If *Fecha Fin De Recurrencia* is set in between the programmed dates of activities of the sequence, the whole sequence must be programmed ***completely***, exceeding if needed *Fecha Fin De Recurrencia.*

**26Jun23:**

**ProgActivities functions.**

**Execution instance:** The programming of the execution of a ProgActivity. An execution instance is defined by the creation of a record in table tblLinkPA which defines: ActivityID, target object and programmed execution date (fldProgrammedDate).

* A ProgActivity can be defined by itself without any execution instances. Those execution instances can be created at a later stage by the user or by the logic of the program.
* Once an execution instance is defined, the closure checking methods will check all performed activities against the conditions of that instance and will close it if conditions are met.
* An execution instance is closed by setting the idClosingActivity field for the ProgActivity in the tblLinkPA table.

**target object:** An object that is the target (the subject) of the execution of a ProgActivity. Can be Animal, Person, Tag or Device.

(Target objects for a given ProgActivity are defined in individual records in the table *[Link XXX Actividades Programadas]* where XXX is Animales, Caravanas, Personas o Dispositivos. All target objects for a given ProgActivity can be found by querying the table above with the ProgActivity ID. All records found will correspond to object to which that ProgActivity applies).

Define the following ProgActivities handling functions in the target object classes (Animal, Tag, etc.):

* *paUpdate()*: a method *to add or remove* ProgActivities from a target object (adding to or removing from its myProgActivites data structure)

Should be invoked whenever there is a change in the target\_object condition or parameters (changes in:

* + - Location
    - Category
    - Age
    - Status
    - Monetary Transaction Performed
    - Ownership modification/change

It should also be called when a new target object is added to the system (alta) or removed from the system (baja)

*The acts of generating a new ProgActivity or removing a ProgActivity from the list due to closure are not covered by update. They are performed by other dedicated methods.*

The idea is the function above is called both by the user and by the system in order to update which progActivities apply to a given target\_object at any given time.

In the case of the system, updateProgActivities will be called by the IntevalTimer threads while these threads sift through the target\_object lists and detect/perform condition changes.

From a design standpoint, for the sake of integrity and modular expansion of the system we must generalize the *supportsPA* flag to execute all ProgActivities-related method after an Activity has been executed. The methods to execute when the *supportsPA* flag is set are:

* **paUpdateList ().** This one executed:
  + By Activity specific methods (set, get, etc). Will use self to determine Activity being executed and look up in RAP all records that have that Activity, and its new value as conditions.
  + Independently (and asynchronously in the background) by IntervalTimer threads that scan the full target\_object list for condition changes. (Category changes, age, timeout changes, etc).
* **paMatchAndClose() (**and inside this one, \_\_checkFinalClose()). *This one executed in an independent, asynchronous thread with the use of a BufferCursor,* ***so must be executed last. Always. And NOT wait for any results from it.***

In the ProgActivities class, define:

* createProgActivity(): Already written. Need to check its coding. In particular, add the checks and updates of the Activity register mentioned above -> When a ProgActivity is created, its Activity name should be inserted in the Activity register so that the system can apply the same logic as paMatchAndClose and execute the updateProgActivities method when the Activity code is executed.

The way pa functions operate as of now is:

* paUpdate(): ADDS ProgActivities to an object, based on the value the self Activity acquires. *-> Perfect candidate to send to a AsyncBuffer for processing, right after \_paMatchAndClose ().*
* matchAndClose: CLOSES and REMOVES ProgActivities for the object.
* IntervalTimer function activity to define (name: paCleanup()???) that walks the target object registry and closes and removes all activities that have expired. They are all closed with ‘*closedNotExecuted’* status.

The paUpdate() method will have to:

1. Traverse the whole list of active records in RAP and:
   1. Check for which records (ProgActivities) the present Activity change is a condition (it may be an Activity with same or a different activityName).
   2. When 1 record found, add the corresponding ProgActivity with myProgActivities.
2. If calling activity (self.activityName) is ‘Alta’, must add ALL active ProgActivities based on:
   * 1. Location
     2. Gender (MF).
     3. Category
     4. Age
     5. Race (if any)

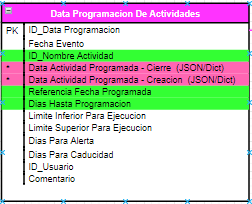
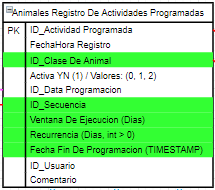
Implement a class (and an Activity in DB Table) named ObjectManitenance. Decorator name: ‘maintenance’. Methods:

* cleanupPA()
* Any other type of update or cleanup to be done in the background.

**10Jul23:**

Updated definitions in [*Data Programación De Actividades*]:

* Referencia Fecha Programada
* Dias Hasta Programación.

**Triggers are discontinued. No more triggers.**

All the functionalities previously assigned to Triggers are now part of ProgActivities (split among the various tables for ProgActivities).

**Conditions vs. Triggers**: Forget all about triggers above***. This is the real breakthrough: Conditions replace Trigger.***

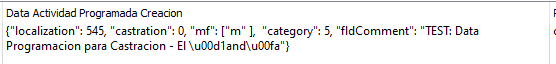
When an Activity executes for a specific target object (called outerObject in the code):

1. Complete execution and update the status of the object attribute. That new value is the value to check against any conditions.
2. Travel the whole RAP table (active ProgActivities) checking for the condition that the Activity just modified. *This will be done using the same name for both: Activity and Condition.*
3. For each ProgActivity found to have the condition modified by the Activity, check if the values match as follows:
   1. There is a match for this condition:
      1. If ALL other conditions are met for ProgActivity:
         1. *create a* ***new execution instance*** *in tblLinkPA for the ProgActivity in question with target object as the object and fldProgrammedDate and other parameters from the ProgActivity object.*
         2. Go to c.
      2. If any of the conditions are not met: Nothing done. Go to c.
   2. There is NO match for this condition: Nothing done. Go to c.
   3. Continue looping: Go to next ProgActivity until all records in RAP are examined.

Along with the introduction of the concept of ***Execution Instance***, other key new concepts are:

Every ProgActivity has 2 very distinct states that require independent sets of parameters:

* **Execution Instance Creation**: The generation of a record in tblLinkPA, for a given Activity, for a specific target object when ALL of the conditions set forth in the Creation dictionary are met.
* The creation of an execution instance can be performed by the user or by the system.
* The creation of an execution instance responds to a change in the state of the target object. Therefore, whenever an Activity is performed, a check must be made as to whether there has been a change in state that triggers the creation of an execution instance. This check is performed by \_paCreateExecInstance() method, run in all applicable Activity set() methods. (\_paCreateExecInstance() is run in the background, using the ProgActivityBuffer class).
* For the creation of an execution instance to take place***, ALL of the conditions in the fldPADataCreacion dictionary must be True***.
* The conditions set in the dictionary are ALL attributes of the target\_object. Any one of them not being an attribute is ignored. Then, a typical dictionary is as follows:



Here, the field *fldComment* is not an attribute of the object, hence it will be ignored.

* **Execution Instance Closure**: The closure is the same as defined above. Checked and performed by paMatchAndClose() method.

The important conceptual change here is the addition of the Creation concept, totally independent from execution instance Closure, and with a set of parameters that is also independent of the closure parameters.

A new logic has been defined to make comparisons for Creation and Closure of ProgActivities.

The logic is implemented via attributes of the objects and functions as follows:

1. Objects implement a *comp()* method to make comparisons of attribute values.
2. When comp() is not implemented for the object, the logic searches for the attribute value itself.
   1. If the type and value of the attribute value and comparison value match, a comparison is performed with a *compare()* function.
   2. If the types don’t match, the comparison is False.
3. Creation compares ALL attributes.
4. Closure compares both attribute and field values.
5. In the Creation of an execution instance, a fldProgrammedDate date value is mandatory: All ProgActivities MUST have a programmed date. In order to achieve this, a set of Reference Dates are defined, so that the Programmed Date can be tied to the conditions that trigger the creation of the execution instance.

For this, the get() methods of all activities return the date of the last state change when called with the key ‘get\_date’

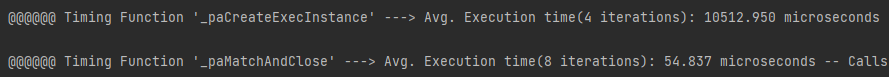
The code to compare attribute values in \_paCreateExecInstance() is

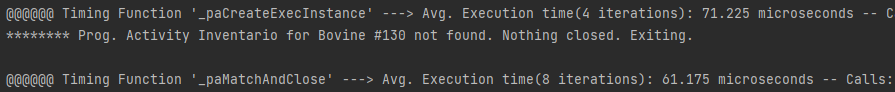
createDict = temp1.unpackItem(fldID=temp.getVal(j, 'fldFK\_DataProgramacion')).get('fldPADataCreacion', {})  
for k in createDict:  
 if k in self.\_excluded\_fields:  
 createDict.pop(k) # para remover 'fldComment', etc.  
# Check conditions defined in createDict against current states and attributes in the target object.  
# *TODO(cmt): All keys in createDict MUST match outer\_obj's attributes (methods, properties or variables).*# *First tries comp(), 2nd tries get() with compare(), 3rd assumes it's a property and also uses compare().*# *If k is not a valid outer\_obj attribute, skips and continues with next k.*for k in createDict:  
 if hasattr(outer\_obj, k.strip().lower()):  
 attr = getattr(outer\_obj, k.lower())  
 else:  
 continue # attribute k not found, goes to next k.  
 try: # executes if conditions[k] implements comp().  
 res = attr.comp(createDict.get(k))  
 except (AttributeError, TypeError, KeyError, ValueError):  
 try: # If comp() is not implemented, tries get()  
 res = compare(attr.get(), createDict.get(k))  
 except (AttributeError, TypeError, KeyError, ValueError):  
 # If get() is not implemented assumes it's a property (ex. dob): uses that value for compare()  
 res = compare(attr, createDict.get(k))  
 matchResults[k] = res

The definition of fldProgrammedDate for the activity is performed by the code below:

addedPA = list(paSet.difference(outer\_obj.myProgActivities)) if paSet else None # paSet not empty -> Hay PA.  
if addedPA: # if there's new progActivities create records in tblLinkPA for outer\_obj, write to DB and register  
 tblLinkPA = DataTable(self.\_\_tblLinkPAName)  
 for j, o in enumerate(addedPA):  
 if isinstance(o.referenceEvent, (int, float)): # TODO: leave this option for now. See if it's of any use.  
 # referenceEvent es el dia del año a asignar a fldProgrammedDate. daysToProgDate debiera ser 0.  
 progDate = datetime(eventDate.year, 1, 1, eventDate.hour, eventDate.minute, eventDate.second) + \  
 timedelta(days=o.referenceEvent)  
 if progDate + timedelta(days=o.daysToProgDate) < eventDate - timedelta(days=o.lowerWindow):  
 progDate = datetime(progDate.year+1, progDate.month, progDate.day, progDate.hour,  
 progDate.minute, progDate.second) # Adds 1 yr.  
 elif isinstance(o.referenceEvent, str):  
 try:  
 # TODO(cmt): o.referenceEvent is type str and outer\_obj.getattr(outer\_obj, o.referenceEvent) is  
 # a property. 3 possibilities to get the fldProgrammedDate reference date (aka progDate):  
 # If str o.referenceEvent converts to datetime, that will be the ref\_date. Else:  
 # If outer\_obj.getattr(outer\_obj, o.referenceEvent) returns a datetime object, that's ref\_date.  
 # Else, if outer\_obj.getattr(outer\_obj, o.referenceEvent) implements get(), calls get().  
 # Else progDate = None.  
 # Finally, if ref date is datetime, progDate = ref date + timedelta(days=o.daysToProgDate)  
 progDate = getattr(outer\_obj, o.referenceEvent)  
 if not isinstance(progDate, datetime): # dob, for instance, will return datetime directly.  
 # if value returned directly is not datetime, attempts to execute get()  
 progDate = getattr(outer\_obj, o.referenceEvent).get(event\_date=True)  
 except (AttributeError, TypeError, ValueError):  
 try:  
 progDate = datetime.strptime(o.progDateRef, fDateTime) # Si es datetime asigna directamente  
 except (TypeError, ValueError):  
 progDate = None  
 else:  
 progDate = None  
  
 if isinstance(progDate, datetime):  
 if isinstance(o.daysToProgDate, (int, float)):  
 progDate += timedelta(days=o.daysToProgDate)  
 if progDate >= eventDate - timedelta(days=o.lowerWindow):  
 tblAux = getRecords(tblLinkPA.tblName,'','',None, '\*', fldFK=outer\_obj.ID, fldFK\_Actividad=o.ID)  
 if tblAux.dataLen:  
 continue # skips if a record for that ProgActivity ID and that target object already exists  
 # 1. Creates execution instance. This record will be repository to get myProgActivities from DB.  
 tblLinkPA.setVal(j, fldFK=outer\_obj.ID, fldFK\_Actividad=o.ID, fldProgrammedDate=progDate,  
 fldComment=f'Activity {o.activityName} created by system on {eventDate}')  
 # 2. Register in PA memory dict  
 outer\_obj.registerProgActivity(o)  
 # Actualiza tblLinkPA con todas las ProgActivities agregadas.  
 if tblLinkPA.dataLen:  
 tblLinkPA.setRecords()

Below the difference between executing all of the code above from \_paCreateExecInstance() by itself (top picture) and executing it in a separate thread, using AsyncBuffers (bottom picture).

****

****

The full description of the logic for comparison and for fldProgrammedDate assignment is detailed in the coding of functions \_paCreateExecInstance() and \_paMatchAndClose() and \_\_isClosingActivity().

1. Closing a ProgActivity is a different story: There are ***parameters of the already executed activity*** that must be matched with the parameters in the Close dictionary of the ProgActivity. So in the case of the closure of an execution instance it’s a question of both *parameters and conditions*.

Important Definitions.

**Condition:**

A state of an object’s internal attributes. It is required to determine on the *Creation* of an execution instance. Also used to determine matches in the *Closure* of an execution instance. Ex.: dob, age, mf, category.

All conditions are defined by the name of the attribute in the target object so that a call to getattr() yields the attribute value required for the comparison.

This is particularly important as it will cover attributes that may be ***methods, properties or variables*** within the object structure.

Parameter.

An element (variable) in the execution instance of an Activity (performed Activity) that may or may not be an attribute of the target object. It can be a database data field value.

Parameters are used in the *Closure* of execution instances (but not in their Creation).

**Recurrence.**

Now Recurrence is handled at ProgActivity Level (it’s an attribute set in RAP) and the variables required to define fldProgrammedDate during the definition of a new ProgActivity are defined in [Data Programacion De Actividades]

**Typical implementation of comp().** In this example, comp() as a method in Geo class to compare 2 localization objects.

def comp(self, val):  
 *"""Returns True if outer\_obj within self is contained (geographically) in val"""* if isinstance(val, int):  
 val = Geo.getObject(val)  
 if isinstance(val, Geo):  
 return self.get().contained\_in(val)  
 return False

**Implementation of compare() and compare\_range() functions.** Defined in krnl\_config.py

def compare(val1, val2):  
 *""" Checks if val1 is equal or is in val2, based on the type of val2.* ***@param*** *val1: single value* ***@param*** *val2: value or structure  
 """* if isinstance(val2, (list, tuple)):  
 try:  
 return val1 in val2  
 except(TypeError, ValueError, AttributeError):  
 return False  
 elif isinstance(val2, str):  
 try:  
 return removeAccents(val1) in removeAccents(val2)  
 except(TypeError, ValueError, AttributeError):  
 return False  
 elif isinstance(val2, set):  
 try:  
 return not(bool(set(val1).difference(val2))) # True si val1 esta incluido (todos sus elementos) en val2.  
 except(TypeError, ValueError, AttributeError):  
 return False  
 elif isinstance(val2, dict):  
 if not isinstance(val1, dict):  
 try:  
 val1 = dict((k, v) for (k, v) in val1)  
 except(TypeError, AttributeError, KeyError, ValueError):  
 return False  
 try:  
 result = [k in val2 and v == val2[k] for (k, v) in val1.items()]  
 return all(j is True for j in result) or val1 == val2  
 except(TypeError, AttributeError, KeyError, ValueError):  
 return False  
 else:  
 return val1 == val2 if type(val1) == type(val2) else False

def compare\_range(comp\_val, ref\_val, low\_val=VOID, high\_val=VOID, \*, exclusive=False):  
 *"""  
 Compares comp\_val to a value with lower and upper limits (Deviation) passed in data\_list.  
 Rules (assumes comp\_val is always passed):  
 1) No values passed -> returns False  
 2) ref\_val != None. All rest is None (1 value passed) -> compares as ==  
 3) ref\_val OR low\_val != None -> ABSOLUTE COMPARISON (NO reference value).  
 3.1) (None, low\_val) -> comp\_val <= low\_val  
 3.2) (ref\_val, None) -> comp\_val >= ref\_val  
 4) ref\_val, low\_val, high\_val != None -> RELATIVE COMPARISON (using reference value).  
 ref\_val -low\_val <= comp\_val <= ref\_val+high\_val* ***@param*** *exclusive: True: use <, >. False: use <=, >=.* ***@return****:  
 """* if comp\_val is None and ref\_val is None:  
 return False  
 if low\_val is VOID and high\_val is VOID: # 1 value passed. Compares as ==.  
 return comp\_val == ref\_val if type(comp\_val) == type(ref\_val) else False  
 elif high\_val is VOID: # 2 valueS passed. Compares as <= or >=.  
 if ref\_val is None: # (comp\_val, None, low\_val) => chequea comp\_val <= low\_val  
 if low\_val is None:  
 return False  
 return comp\_val < low\_val if exclusive else comp\_val <= low\_val  
 elif low\_val is None: # (comp\_val, ref\_val, None) => chequea comp\_val >= ref\_val  
 if comp\_val is None:  
 return False  
 return comp\_val > ref\_val if exclusive else comp\_val >= ref\_val  
 else:  
 if comp\_val is None:  
 return False  
 try:  
 low\_dev = min(ref\_val, low\_val)  
 high\_dev = max(ref\_val, low\_val)  
 except (TypeError, ValueError):  
 return False  
 return low\_dev < comp\_val < high\_dev if exclusive else low\_dev <= comp\_val <= high\_dev  
 else: # All 3 values passed. Compares as ref-low <= comp\_val <= ref+high.  
 if any(j is None for j in (ref\_val, low\_val, high\_val)):  
 return False  
 try:  
 low\_dev = ref\_val - low\_val or 0  
 high\_dev = ref\_val + high\_val or 0  
 except (TypeError, ValueError):  
 return False  
 return low\_dev < comp\_val < high\_dev if exclusive else low\_dev <= comp\_val <= high\_dev

**20Jul23. Differed processing.**

**Differed processing** is the delayed execution of a function/method that processes objects or data. The execution is asynchronous in the sense that the time of execution is determined by the system, based on different timing criteria.

This is accomplished by “enqueuing” the process to be differed in a dedicated queue (usually in a separate, independent thread) and having a queue managing application that runs in the background and that executes the function at a later time, freeing the method that calls the differed function almost immediately.

The main use for this is free-up the front end or other critical threads by queuing resource intensive tasks to be executed in the background.

The differed methods may or may not return values. *However, if they return any data, all the advantages of the differ/postponement are lost as the caller function has to wait for a result.*

The queue managing logic uses classes:

* *AsyncBuffer* as the buffer manager that starts a separate thread, creates the storing structure (buffer object) and runs the queue monitoring process.
* *BufferAsyncCursor* as a cursor object class that contains the differed function information (function object itself, calling object and function arguments).
* *Writer:* a class that implements the queue handling details, including the queue-checking routine and execute function that launches the differed function code.

**Implementation**: Differed processing is implemented with the use of a @decorator defined for the functions that are to executed the differed tasks.

At the time of writing there are 2 differed methods (both in Activity Class) and used to process Programmed Activities creation and closure in the background:

* *\_paCreateExecutionInstance():* Creates a new ProgActivity. Updates related memory structures.
* *\_paMatchAndClose():* Verifies if a given executed Activity qualifies as a Closure Activity for a ProgActivity. If so, closes the ProgActivity and updates memory data structures and related database tables.

The object for which the differed method is executed (outerObject) must be passed as an argument to the differed method since outerObject is implemented as a dynamic, thread-dependent argument (each thread keeps one internal, independent copy of outerObject so switching threads will switch the outerObject object). This is accomplished in 2 ways:

1. outerObject is passed in function kwargs: the passed object is used.
2. outerObject is not passed in function kwargs: outerObject is assigned the activity’s outerObject value and passed to the function in kwargs.

The decorator method used to enqueue these methods in the differed processing buffer is shown below:

@staticmethod # *Passing buffer as an arg. Cannot define the buffer inside because of buffer management.*def \_\_paDifferedProcessing(async\_buffer=None): # This wrapper works wonders in a separate, prioritized thread!  
 buffer = async\_buffer  
  
 def \_\_aux\_wrapper(func): # This wrapper is needed in order to pass argument async\_buffer.  
 @functools.wraps(func)  
 def wrapper(self, \*args, \*\*kwargs): # self is an Activity object.  
 # enqueue() is the only way to access the cursor object for the call. Results are NOT pulled by default.  
 # Only if the buffer queue is running.  
 if not buffer.is\_stopped: # Accesses \_is\_stopped directly to minimize use of locks.  
 # print(f'\nlalala \_paMatchAndClose({lineNum()})!! : self:{self} /args: {args} / kwargs: {kwargs}'  
 # f' / outer\_obj\_id: {self.outerObject.ID} ')  
 """ MUST pass outerObject for the thread because \_\_outerAttr is dynamic & thread-dependent. """  
 if not kwargs.get('outer\_obj'): # if outer\_obj is passed in \_paMatchAndClose kwargs, keeps it.  
 kwargs.update({'outer\_obj': self.outerObject}) # If not, pulls it from outerObject.  
 cur = buffer.enqueue(\*args, the\_object=self, the\_callable=func, \*\*kwargs) # *TODO:TO ANOTHER THREAD!* # *TODO(cmt): Si espera por cur.result, se pierde toda la ganancia (50 usec va a 3 - 12 msec!)* if cur and kwargs.get('get\_result') is True: # Reserved feature:wait and return results if required  
 return cur.result  
 return None # Normal use:return None and not wait for the processing of the cursor by writer thread  
 return func(self, \*args, \*\*kwargs) # Si no hay AsyncBuffer, ejecuta normalmente.  
 return wrapper  
 return \_\_aux\_wrapper

**Then,** a differed method is implemented as follows:

@\_\_paDifferedProcessing(\_\_progActivityBuffer)  
def \_paMatchAndClose(self, idClosingActivity=None, \*args, outer\_obj=None, execute\_fields=None, excluded\_fields=None, closing\_status=None, force\_close=False):

\_\_*progActivityBuffer* is an object of class AsyncBuffer that handles all then queuing and postponed processing for \_paMatchAndClose() method.

**11Sep23. Field Name matching and more.**

Typical executed Activity dictionary passed to \_\_isClosingActivity()

***execute\_fields*** *= {'execution\_date': datetime.datetime(2023, 12, 15, 15, 58, 14, 929020), 'fldFK\_ClaseDeAnimal': 1, 'age': None, 'localization': None}*

Typical ProgActivity dictionary passed to \_\_isClosingActivity()

***paFields*** *= {'fldProgrammedDate': datetime.datetime(2023, 11, 10, 15, 55, 39, 387033), 'fldWindowLowerLimit': 30, 'fldWindowUpperLimit': 30, 'fldDaysToAlert': 5, 'fldDaysToExpire': 60, 'fldPAData': {'age': [180, None], 'fldComment': 'Test Inventario- El Ñandú', 'fldMF': 'm', 'localization': '42990b601cec4ddb9d85bfb94cda2e29'}, 'fldFK\_ClaseDeAnimal': 1, 'fldFK\_Secuencia': None}*

Note that:

1. “Non-fld” keys (category, age, execution\_date, etc) can and do appear in executed Activity parameters dict.
2. Also, “Non-fld” keys (category, age, localization, etc) can and do appear in ProgActivity closing parameters dict.

Typical d2Flat (flattened dict) from paFields dict.

***d2Flat*** *= [{'fldProgrammedDate': datetime.datetime(2023, 11, 10, 15, 55, 39, 387033)}, {'fldWindowLowerLimit': 30}, {'fldWindowUpperLimit': 30}, {'fldDaysToAlert': 5}, {'fldDaysToExpire': 60}, {'age': [180, None]}, {'fldComment': 'Test Inventario- El Ñandú'}, {'fldMF': 'm'}, {'localization': '42990b601cec4ddb9d85bfb94cda2e29'}, {'fldFK\_ClaseDeAnimal': 1}, {'fldFK\_Secuencia': None}]*

The iterator function creates a list of 1-key dictionaries.

**Rules for Field Name matching.**

When comparing field names in \_\_isClosingActivity() method, the following rules apply.

1. Comparison is done between data generated by the executed Activity that acts as the “closing” activity (execute\_fields dictionary) and ProgActivity data (paData dictionary) that is stored in the database ProgActivity tables.
2. ***proper field name***: a string in the form “tblName.fldName”.

Proper field names are used when the comparison required is between 2 related fields with field names are different and are NOT a subset of the other (Ex*.: tbl1.fldID* is to be matched with *tbl2.fldFK\_Categoria*).

1. When a proper field name is found, it can be validated only with another proper field name. This process uses function getFldCompare().
2. ***special name***: A string that *starts with or is equal to its associated shortName* and does NOT have the particles “fld” or “tbl” in it. It is usually equal to the Activity object’s property name. Example: category, inventory, age, execution\_date, status.
3. ***shortName***: An attribute of Activity objects that is comprised of *all the characters of the Activity special name, up to the 6th character*. It also must be a substring of the field names it is to be matched to.

It is equal to the property name if the property name length is <= 6.

It is used to match field names with special names like age, localization, status, category, etc.

**All** Activity objects have shortName defined when they have a property name associated to them.

A general rule stems from this definition:

***All property names must be such that their first 6 characters are a substring of the field names they are associated with.***

Examples (property name -> shortName):

status -> status, localization -> locali, category -> catego, inventory -> invent

1. When a key string in execute\_fields dictionary contains the particle “***fld***”:
   1. If the key contains “.” checks for proper field names using getFldCompare() and returns the result from the call.
   2. If “.” not in key string (but “fld” is), searches for key in the paData dictionary.
   3. If no matches with key, derives shortNames from all of the keys in paData and checks of there’s a match with shortNames.
2. When a key string in execute\_data dictionary does not contain the particle “***fld***”:
   1. Pulls shortName for key and checks the shortName against each of the keys in paData.
   2. Sdfsd

With the above rules, the matching map between execute\_fields and paData performed by \_\_isClosingActivity() is below:



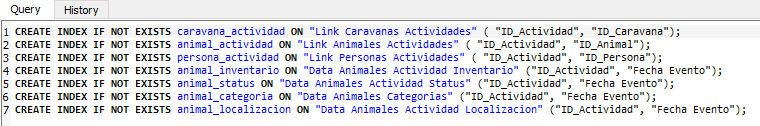
**20Sep23. Indices.**

Indices will be implemented for all Link XXX Actividades tables. The objective is to speed up queries and searches using JOIN statements via the \_getRecordsLinkTables() function since these JOIN queries are used in all the get() calls for any Activity and Object.

Additionally, indices for the most frequently used Actividad Tables are created (Actividad Inventario, Actividad Categoria, Actividad Localizacion, etc.) in order to optimize the access with a given ID\_Actividad and the last recorded date (Fecha Evento) which is the most commonly used query in these tables.

*Note: Indices are used automatically and behind the scenes by SQLite. No need to do anything about them other than defining it correctly for their intended use.*

At the time, 3 implemented in database version 4.1.8:

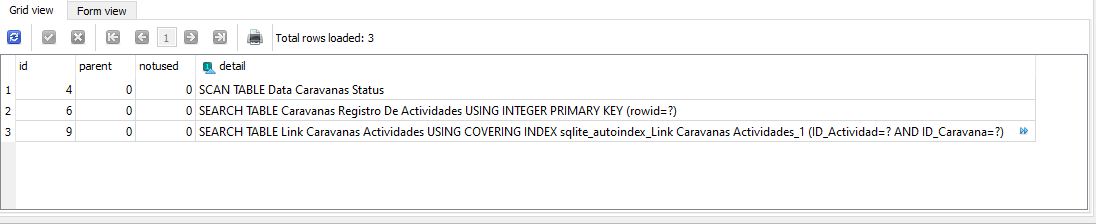


To validate the use of these indices, the QUERY PLAN below

*EXPLAIN QUERY PLAN*

*SELECT "Data Caravanas Status".\* FROM "Caravanas Registro De Actividades" INNER JOIN "Link Caravanas Actividades" USING ("ID\_Actividad") INNER JOIN "Data Caravanas Status" USING ("ID\_Actividad") WHERE "Link Caravanas Actividades"."ID\_Caravana"="8e9005df8ee443019ff6422f538191c4"*

yields:



**18-Dec-2023. Processing of external RA records in the closure of Programmed Activities.**

Performed (executed) Activities that are completed by other nodes are replicated eventually across all nodes.

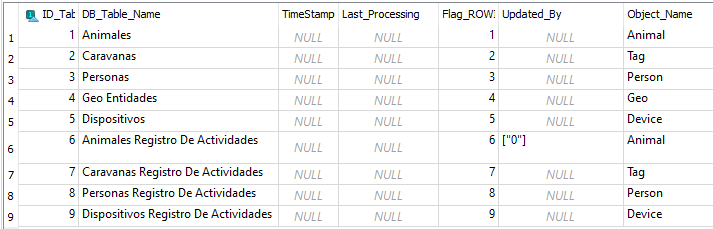
The completion of these activities may trigger closure of ProgActivities defined in a node.

This is managed on a node-by-node basis with the following criterion:

RAP external records will be matched only with local ProgActivities records in search for closure conditions and any matching ProgActivity Records will be closed in that node. *Nodes close ONLY their local ProgActivity records*. Each node takes care of their local ProgActivity records for the sake of simplicity and consistency. All these “local” closures will eventually be replicated to the rest of the nodes in the system.

In order to manage history of the processing of external RA records, use a TimeStamp for each RA table, stored in \_sys\_Trigger\_Tables, along with the RA table name. Then, the pair (tblName, TimeStamp) can univocally define hat last external record that has been processed by the local node.

The values are already set in \_sys\_Trigger\_Tables and are used for detection of incoming replicated records for specific tables.



Use these values to determine the start point (Last\_Processing) for records to process in function

processExecutedActivity().

In the case of Animales, the Last\_Processing is for ALL Animal classes (Bovine, Caprine, etc), so the RA processing routine must be executed for ALL classes that have newer external records replicated onto the node. The classes that generate new records are listed in the column *Updated\_By,* so the treatment is straightforward.

By the way, *Updated\_By* should not be needed for the rest of the rows. Just Last Processing

**19-Dec-2023: Final, Final words on Triggers:**

**Below is how the triggers end up. All the above is the step-by-step journey but many concepts and definitions may be obsolete.**

**These triggers need to be improved later, replacing the table\_name comparison with the trigger\_name comparison. Tables can be repeated while trigger\_names are unique.**

**DBTrigger class:** A DBTrigger Class is defined in krnl\_db\_access.py to manage all SQLite Trigger in the custom\_types python module.

**24-Dec-2023. Managing duplication. Repeat and One-time activities.**

As duplicated records are produced, they will bring other data associated with them, mainly in the form of Activity records.

2 types of activities are defined that have different treatment of duplication of records.

* **Repeat Activities**: can be executed multiple times for the same object in the lifespan of the object (ex.: Vaccination, certain surgeries, inventory, localization).

By definition it is not practically possible to identify duplicate activities for these activities, so all repeat activities will the treated as distinct, individual activities performed on the object.

* **One-time activities**: by their nature can be performed only once for each object: alta, baja, ventas, castration, destete, paricion (within a certain timeframe).

For these One-time activities, the record to be used in the case of duplication has to be specifically determined for the type of activity. Each activity will have to define a variable to store this information.

In particular:

* + **Using the 1st (earliest TimeStamp) duplicate records:** entrada/alta, salida/baja (in all its forms: sale, lost, death, etc.), weaning (destete), delivery (within a 8 month time span from each other. The earliest one will be considered the valid delivery date for the object, if several delivery activities were recorded for the same object within the defined timespan).
  + **Using the last (latest TimeStamp) duplicate records**:
* Sdfsd

**Chapter 4: Logging**

25Apr23

Logging implementation has been completed with a scheme of 3 loggers:

***fe\_logger***: Front end Logger. Used by all Front-End threads.

***bkgd\_logger***: Backend Logger used by all IntervalTimer functions running tasks in the background (independently from the Front End)

***db\_logger***: Logger used by the modules and functions directly accessing the Database, namely, *krnl\_sqlite.py and async\_cursor.py*

The function krnl\_cfg\_items.getLogger () implements all the details to initialize and launch the loggers. All 3 loggers mentioned above are defined in the krnl\_cfg\_items module.

The print() function has been decorated with a function that checks whether to print or not.

This is done via parameter *dismiss\_print*. This is to ease, simplify and continue to use all the print commands already punched in across the code.

orig\_print = builtins.print #Renombra funcion print()original(en builtins.py)

@printWrapper  
def print(\*args, dismiss\_print=False, \*\*kwargs):

# Definicion de print() que usan TODOS los calls a print() del sistema  
 if dismiss\_print is True:  
 return None # Porque builtins.print() retorna None.  
 return orig\_print(\*args, \*\*kwargs)

Each logger created can be programmed to generate 2 additional files:

* debug = True -> Creates a file NNN\_debug.log with debug level info
* error = True -> Creates a log file NNN\_error.log with entries with level ERROR, CRITICAL.

**Chapter 5: IntervalTimer function and activities (executed in the background).**

**20 April, 2023**

krnl\_threading.py has been re-written, adding class IntervalFunctions to manage and synchronize launching of background functions. The main design criteria are:

* Intervals predefined and arranged in bins/ranges as follows:
* \_\_startSeconds = {\_\_2SEC: (1.5, 2.7), \_\_10SEC: (15, 30, 45), \_\_1MIN: (10, 20, 40, 50), \_\_1HR: (0, 30\*60, 10\*60, 40\*60, 20\*60, 50\*60)}

Here each tuple defines the starting second of a thread. This is done to minimize threads stepping onto each other.

* Number of independent threads in each range are limited to the quantities shown above. Any attempt to launch more threads than allowed returns without launching the thread.

The schedule () function in IntervalTimer class organizes the launch of background threads according with this logic.

The functions being launched in each thread are managed by the methods in class IntervalFunctions.

Also, an object factory is implemented to create IntervalFunction objects. So in order to launch function(s) in a background thread, the factory method must be called with the required parametes. Example:

dailyFunctions = IntervalFuncsFactory.create\_object(interval=0.25 if USE\_DAYS\_MULT else 24\*3600, func\_list=(animalUpdates,), start\_hour=0, start\_thread=False)

**Chapter 6: Animal Sanitation (Sanidad)**

Sanitation is an abstract class from which specific classes are to be derived.

Not clear at this time (May-2024) what form a Sanitation object will take. For now, only functions and methods will be defined.

**Chapter 7: Geo**

**24-Jun-23**

Since v4.1.7 Geo is implemented by the definition of Geo objects, created during system initialization.

A \_init() method reads all rows from table [Geo Entidades] that are in valid state (fldFlag=1) and creates a Geo object for each one of the loaded rows.

Then registers each object in a \_\_registerDict dictionary.

These are all the Geo objects available the system as a whole.

All the attributes and methods for the class are defined in *krnl\_geo\_new.py*.

**Containers.**

Each Geo Entity has a \_\_containers field defined. The field can be a single Geo object or a list of Geo objects that represent the closest immediate Localization Levels that contains the entity. It will be 1 object in most of the cases but when a Geo entity spans across several objects, the \_\_containers field will be a list of Geo objects (Establecimiento that contained in several Departments, or Provinces, for example).

\_\_containers will NOT include self.

**Container Tree**.

Each Geo Entity has a \_\_container\_tree attribute that is a list of all the Geo Objects which *contain the container*s for the entity, including the Geo entity itself, up to the Country level. This attribute is required in order to verify belonging.

\_\_container\_tree **INCLUDES** self.

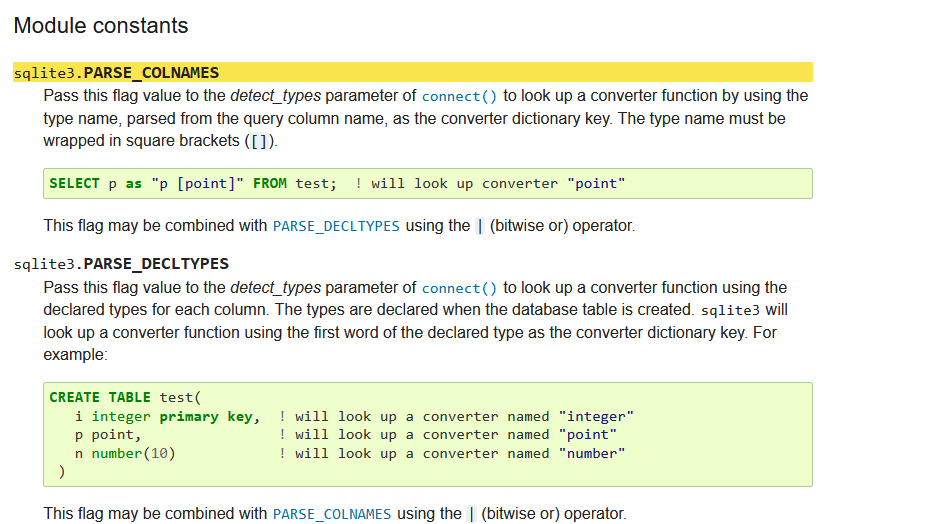
**Localization**.

Localization is an attribute assigned to system objects like Animal, Person, Device or Tag. It is *one and only one Geo object* by definition (as any of the objects listed above can only be located in one place at a time).

Being just one object, the Localization field is turned into a GEO field in SQLite in order to use sqlite3 converter functions (that convert from int to Geo and vice versa) to access the attribute transparently as is being done with TIMESAMP and JSON.

**Note.** When dealing with adapters/converters, COLNAMES cannot be used in the declaration of special types because of the issue below: As the type name for PARSE\_COLNAMES must be wrapped in brackets 🡪 “p as [point]”, that would require special consideration in the parsing functions for getRecords/setRecord/delRecord. So forget about it.

Use only PARSE\_DECLTYPES.



**Chapter 8. Money, Monetary Transactions.**

13Jul23: Monetary Transactions is re-written from scratch mainly to drop the TM handler concept and adopt full TM objects (similar go Geo objects above).

These TM objects are mean to be *volatile* objects used to process monetary transactions and activities, record data in the data base o r read data from database. However, they are not supposed to be registered in dictionaries of kept in registers any longer that required. Volatile in nature.

Another important feature is the adoption of Money objects, from the money module with 2 main attributes: amount and currency. These objects can be processed in whatever way with ease, using the methods, overloads and attributes defined in the Money class.

Additionally, XMoney is a subclass that allows for direct operations among Money objects of different currencies. Very convenient.

With these changes, a full-fledged TM Activity Class will have to be defined to processes activities like transaction, adjustment, amount, etc.

The TM class is generalized to a class *MoneyActivity*. Totally analogous to AnimalActivity, TagActivity, etc.

Derived from MoneyActivity will run the specific activities involving Money: Adjustment, Transaction, RevenueRecognition, RevenueBilling, RevenueCollection, DebtRecognition, DebtBilling, DebtCancellation, LoanTaking, LoanCeding, etc.

Dealing with Decimal type and sqlite.

<https://stackoverflow.com/questions/6319409/how-to-convert-python-decimal-to-sqlite-numeric>

From what was experimented with Decimals:

1. Decimal objects (*amount* attribute in Money objects) will generate an approximation in value. Ex: 12.490 turns to 12.49000000000001087130385712953284382820129394531250.
2. Importantly:
   1. if the Money instances is created with a *float,* as Money(12.490, ‘USD’) then that approximate value is written to DB as is.
   2. However, if the Money instances is created with str, as Money(‘12.490’, ‘USD’) then the approximated value will be used internally in the object but the value written to DB will be the correct value 12.490.
   3. As a result of a and b:
      1. Decimals will always carry approximation errors and the operations between them will compound the errors. These errors are usually acceptably small.
      2. *To avoid generating additional rounding errors when writing to database, always create Money objects using an amount of* ***type str.***

15Jul23:

Amount class design aspects:

* *Amounts* class will NOT for now carry description, fldComment etc. That will be left to the Activities themselves in line with the design of Activities for other classes. This includes the design of ProgActivities for MoneyActivities.

*The idea is that an Amount instance is created only with the parameters required to create a Money instance (amount and currency)*. Nothing else.

However, it must carry for now ID in order to facilitate operations and conversions for database storage and retrieval.

* Amount inherits from Money and re-declares all the necessary operators. Uses ALL operators defined in Money class.
* NOT ALL Amount OBJECTS will have an ID and will be written to DB. Only those required will get written to DB.

In particular Amount objects that are result of operations between 2 or more Amount instances will generate new Amount objects with \_\_ID=None. The MoneyActivity functions will be responsible for storing those in DB or not (Once an object is sent to DB, an \_\_ID is generated automatically for it).

**Things to store in Server only:**

Database table with historical exchange rate data: table with daily rates for all currencies in the world. Devices can consult this table over a connection link when needed*. This would be a fallback option* as it costs nothing to implement. The 1st option will be to pull any historical data from some provider in the internet (XE api or similar. See review below)

<https://www.abstractapi.com/guides/best-currency-exchange-apis#best-currency-exchange-apis>

**Important: Must provide support for cryptos!!!**

**18Jul23:**

**Definitions.**

The Money/Monetary Transaction area will operate with the following definitions:

**Activity (equivalent to Operation):** A high-level Action (sale, purchase, price, status) that may involve one or more Transactions.

**Transaction (Activity):** Specific type of Activity used to manage Transaction Objects. In particular, define transfer/changes of ownership of all objects defined in the system.

**Amount (Class):** Storage unit for an Amount (Money object + ID). Used by ***Transaction and Price*** Activities.

**Transaction (Class):** Class derived from EntityObject used to store Transaction and Amount data and to manage the actions defined for them.

With these definitions:

* 1 Activity (Operation) can include multiple Transactions.
* 1 Transaction can include multiple Amount objects.

In most cases, 1 Activity will correlate with only 1 Transaction and 1 Amount. But the architecture is left expressed to implement the system with all the complexity required.

Ex.: This system will handle 1 purchase operation for 47 animals. 47 Transactions are generated under the same ID\_Actividad (1 for each Animal).

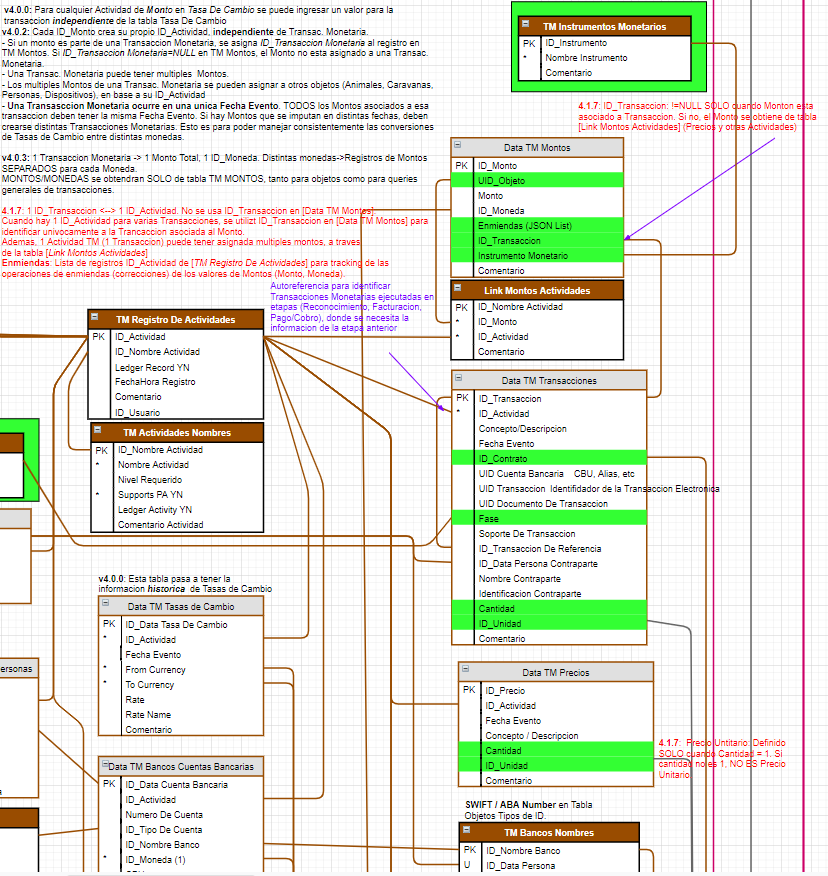
In turn, each Transaction defines 2 or more Amounts (Montos) to express the payment for *each individual Animal* in such species as: portion 1 in ARS, paid with a check; portion 2 paid in USD bills, portion 3 paid with BTC.

Then, the general rule is**: *In general (but not always)*** *for every Activity there will be a 1 – 1 relation with Transactions, and that Activity will carry any number of Amount objects (Montos records), stored in table [Link Montos Actividades].*

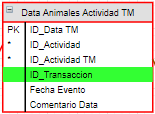
*This is the same design principle as the rest of the Activities / Link Tables / Objects defined in the system:* ***1 Activity applied to multiple objects via the use of a Link table.***

In the particular case of Money Transactions, the equivalent to ‘objects’ above is Amounts, which in nature can be several (more than 1) for 1 Transaction, so the need arises to enable multiple Amounts for 1 Transaction in the system. And given more than 1 Transaction can comprise an Activity, an asymmetry arises versus the other objects in the system that is resolved by adding a field *ID\_Transaccion* in table *[TM Montos]* and in tables *[Data Animales Actividad TM], [Data Personas Actividad TM] and similar.*

The whole system is then expressed as shown below



And the tables linking TM to the different objects must include the additional *ID\_Transaccion field*:



*1: N --> 1 TM Activity : MULTIPLE TM Transactions.  
1: 1 --> 1 TM Transaction : 1 target\_obj (target\_obj is Animal, Tag, Person)  
1: N --> 1 TM Transaction : MULTIPLE Amount objects.*

The above relations are design-critical because of the need to assign 1 or multiple amounts to single target objects (Animal, Person, Device, etc.): If 1 Activity is created for multiple target objects (Sale or Purchase of 25 Animals, for instance) then there must be a mechanism to assign 1 or several Amounts to each individual target object.

This is achieved by defining 1 Transaction for each target object (relation 1:1). Then each transaction can be assigned to multiple Amounts, as needed.

With this in mind, Price is defined as a Transaction so that target objects need to store only the transaction ID in their TM Tables, as shown below (*ID\_Actividad TM* is deprecated).



**Chapter 9: Notifications**

Notifications must make use of NATS JetStream and Marmot replication concepts as much as possible.

**Chapter 10: Dynamic modules.**

Modules can be loaded dynamically into the system, without the need of system re-start.

The design criterion is that modules are loaded dynamically out of a dedicated table in the \_nr database **(\_sys\_Modules** table). The import / load command is executed based on the db record instruction set by Bitmask field.

The \_sys\_Modules table holds records that define module name and module attributes to be loaded. Attribute can include classes, variables, functions etc.

All this is implemented by the **import\_module**() function. The function updates 2 global variables:

MODULES\_DICT, ATTRIBUTES\_DICT

And returns the ATTRIBUTE\_DICT dictionary.

The import\_modules() function is below (from initializer. py module):

def module\_loader():  
 *""" Loads modules defined in table \_sys\_Modules (\_nr database) and creates a dictionary with all attributes defined  
 in those modules, by scanning all rows in the table and importing/reloading records with import or reload bits set.  
 Each time a module is loaded MODULES\_LIST, ATTRIBUTES\_DICT are updated with the new modules and the attribute  
 objects defined therein.  
 If the RELOAD bit is set in Bitmask field, reload is forced on the the module.* ***@return****: Attributes dict {attr\_name: attr\_object, }; ERR\_ string on database access error.  
 """* import\_bit = 0 # bit0 from Bitmask: import module (if not in the system) or use module loaded in [sys.modules].  
 reload\_bit = 1 # bit1 from Bitmask: force module reload.  
 temp = dbRead('tbl\_sys\_Modules', 'SELECT \* FROM \_sys\_Modules; ', db\_name=NR\_DB\_NAME)  
 if isinstance(temp, str):  
 return f'ERR\_DBAccess: cannot read from table tbl\_sys\_Modules. Error: {temp}.'  
 if temp: # goes to work only if temp table holds 1 or more records (temp.\_dataList is not empty).  
 cols = temp.getCols('fldModule\_Name', 'fldAttributes', 'fldBitmask')  
 mod\_attr\_dict = dict(zip(cols[0], cols[1])) # mod\_attr\_dict = {module\_name: [attr\_name1, attr\_name2, ], }  
 bitmask\_dict = dict(zip(cols[0], cols[2])) # bitmask\_dict = {module\_name: bitmask(int), }  
 module = None # marker to signal that a module has been loaded.  
 for idx, k in enumerate(mod\_attr\_dict): # Must enumerate to access fldID to reset reload bit in Bitmask.  
 if test\_bit(bitmask\_dict[k], reload\_bit):  
 try: # 1st tests for reload bit.  
 if inspect.ismodule(MODULES\_DICT.get(k)):  
 module = importlib.reload(MODULES\_DICT.get(k))  
 krnl\_logger.info(f'Module {k} reloaded.')  
 else:  
 module = importlib.import\_module(k) # If module was not loaded beforehand, imports it.  
 except (ModuleNotFoundError, ImportError):  
 continue # On failure, skips to the next module name (keys in mod\_attr\_dict).  
 elif test\_bit(bitmask\_dict[k], import\_bit):  
 try: # 2nd tests for import bit.  
 module = importlib.import\_module(k)  
 except (ModuleNotFoundError, ImportError):  
 continue # On failure, skips to the next module name (keys in mod\_attr\_dict).  
 else:  
 continue # If neither bit is set, moves on to the next module listed in mod\_attr\_dict.  
  
 if module is not None:  
 MODULES\_DICT.update({k: module}) # appends the module object to the master dict for imported modules.  
 # j is each attribute\_name defined in the JSON Attribute field in \_sys\_Modules, in \_nr database.  
 ATTRIBUTES\_DICT.update({j: getattr(module, j) for j in mod\_attr\_dict[k]}) # updates master attribs dict  
 # Resets import and reload bits after import is complete. Saves in \_sys\_Modules table.  
 setRecord(temp.tblName, fldID=temp.getVal(idx, 'fldID'),  
 fldBitmask=bitmask\_dict[k] & ~(1 << reload\_bit | 1 << import\_bit), db\_name=temp.dbName)  
 print(f'Imported MODULES: {MODULES\_DICT}')  
  
 return ATTRIBUTES\_DICT

The criteria also are such that importing attributes from modules is a last resort option, when new variables need to be introduced.

The particular case of subclassing is to be handled by code within the classes (parent and child classes) since the classes are defined in the module: the code in the class definition executes updating all required structures within the classes and avoiding as much as possible the definition of new attributes that need to be imported by the module. A good example of this idea is implemented with the sub-classing of the Tag class when defining subclasses for newly-added Tag technologies.

This approach also to be used when sub-classing the Device class to accommodate for miscellaneous devices like tag readers, geo-locators, sensor/transducer readers, etc.

**Chapter 10: Memory Data.**

**General Concept:** Memory data are data structures stored in memory for ease of access. They are populated by database information and the data stored in memory must mirror at all times the data stored in the database.

The idea of Memory Data (mem\_data) was to be implemented for Activities that handle certain type of data that is often required. In particular: Inventory data, Status data, Category data (for Animals) and Localization data.

The criterion is to keep the data in memory for oft-required Activities so that

* Database access
* The creation of Entity Objects (Animal objects, Tags, Device, etc)

Is minimized.

For this:

1. Each Entity Object class keeps a “general” dictionary that is loaded start up and is kept alive with additions and removals by dedicated functions. This dictionary stores all active objects for the class (objects that pass the isValid() criteria) -> For later on, see the need to create a non\_active objects dictionary).

This dictionary exists from before and is used to manage the object duplication and replication logic.

active\_uids\_dict = {uid, \_duplicationIdex}

1. A dedicated dictionary is created as an attribute of all classes that require mem\_data. It is of the form

\_local\_uids\_dict = {uid: DataMemory object, }

The logic to manage this dictionary (keep it synchronized with the master dictionary described in point 1) is described above in the summary notes.

The DataMemory class is defined as an inner class, inside class Activity. It is made of a internal dictionary that stores specific databases fields from 1 or more db tables.

1. Initialization.

Each class initializes its data\_mem structure (dict) via EntityObject.\_\_init\_subclass\_\_() method.

There, data is

1. Ssdfsdf
2. Sdfsdf

**Initialization and update of Memory Data**

**Initialization** happens during startup by creating memory dictionary structures and objects for each Activity that supports memory data.

The design is such no other actions are required from there on to keep mem\_data structures up to date.

That is, **updates** to dictionaries (additions, removals) are performed by the mem\_data\_get() / mem\_data\_set () functions, whenever new objects are added to or removed from the system.

**Chapter 11: Tags.**

Tags are conceptually one single type of object (one class), however they will be sub-classed based in the db field fldAssignedToClass, from tblCaravanas.

**This is to run smaller data structures and make the code more efficient.**

Classes for different EntityObjects will be: TagBovine, TagCaprine, TagDeviceMachine, TagDeviceVehicle, etc.

If a record in tblCaranvanas is not assigned to an object 🡪 see what to do. In principle, and to be consistent with the general structure of the system, Tag class should be Abstract.

**Chapter 12: Memory Data**

**24Jul24:** Memory Data for Activities will be implemented with pd.DataFrame objects: 1 DataFrame per Activity implementing mem\_data.

Each row in the dataframe is the status of an specific object. The 1st column of the dataframe is the object’s uid.

***Development notes continued in Tag Lifecyle, Tag identifiers.docx.***