**Tag lifecycle, identifiers and duplication management**

New concept (implementation. The concept is not new): **Physical Attachment.**

Physical attachment is the physical association of a tag to an entity (Animal, Device, etc.).

It consists in a physical, verifiable action performed by a user on the entity object: adhering or attaching a physical tag to the object, by which the object will be univocally identified onwards.

In the system, the physical attachment action is called **Commission** of the tag. It identifies the point in time after which the tag identifier is to be used for identifying the entity object.

Attributes:

* Physical attachment (Commission) is the 1st action a Tag is assigned in the system. Any action for that tag that happens earlier in time is considered an input error for that tag.

Due to the distributed, unsync’d nature of the nodes implementation, there may be operations on a certain node that are recorded for a tag that is not defined there (that is, the original tag record, with its commissioning data has not been yet replicated to the node). This attribute demands that those operations happen *later* in time than the time the Commission activity is recorded.

* The Commission Activity is recorded in a dedicated field in the Caravanas table (fldCommission), which is comprised of 2 flags for now:
  + Commission (bit 0): The tag identifier has been assigned a Commission operation
  + ConfirmedByUser (bit 1): During the selection of multiple options, this tag record has been validated by the user as the correct one.

This field is used by SQLITE Caravanas INSERT Trigger to update the \_Duplication\_Index value for the Tag following the \_Duplication\_Index resolution logic.

* The \_Duplication\_Index resolution logic will include the use of the fldCommision bits to contemplate the priorities of *Commissioning* and *ConfirmedByUser* conditions, which will take precedence over the existing resolution logic that picks the record with the earliest creation date.

*ConfirmedByUser as of today is bit 1 in the fldPhyiscallyAttached field. Not discriminated at the moment. If set, it will be treated as a regular PHATT Activity, and the record will be treated as a normal PHATT record for all purposes.*

An important design criterion for PHATT field in Caravanas table is that it is only set to 1 by the *commission.set()* function in Python, and no other action is required in the Python code after that. All the logic is implemented in SQLite.

The only additional requirement is to process PHATT in Tag.\_get\_duplicate\_uids() method, to disregard any Inventory data for uids with PHATT field set to 1.

Note: In the current implementation (using a JSON List of *Identifiers* field in the Animales tables), when an Animal or Device is assigned a tag, the tag’s UID goes to the *Identifiers* field in the object’s record. As the tag \_Duplication\_Index value change (due to data replication), the latest (valid) value of the \_Duplication\_Index will always be accessible via the tag’s UID, which is stored in the objects *Identifiers* col.

Note: The Commission Activity in Caravanas Activities MUST set the PHATT (Physically Attached) bit (bit0) in Caravanas.fldPhysicallyAttached when the Activity is performed, so that the SQL TRIGGER Logic updates all \_Duplicate\_Index fields for a given “Identifiers\_str” records, when a Commission is performed.

Once PHATT is set for the first time, further Commission/Decommission cycles must NOT modify the PHATT bit.

Other points:

* *On the use of Tag Color as one of the Tag Identifier’s elements*: Tag Color, being a user-input value for standard tags, is highly susceptible for input error (subtle difference in color tones may yield incorrect tag identifier strings).

*Because of this, for the time being, Tag Color will NOT be included in the identifers’ strings.*

Very important logic point on Replication (04Aug24):

Marmot implements “eventually consistent” replication.

*However, the system does NOT need to wait for full replication.*

In the case of Animals, Tags, the replication is resolved once the PHATT record is replicated to a node. With the PHATT record, the node has the *original* object record and will always refer to it onwards.

*Take this into account when writing the logic for Bovine.getObject(), Tag.getObject() methods.*

SQLite Duplication Index Logic.

Caravanas INSERT Trigger. What the old trigger does:

CREATE TRIGGER IF NOT EXISTS "Trigger\_Caravanas\_INSERT" AFTER INSERT ON "Caravanas" FOR EACH ROW BEGIN

1. *Sets values for Terminal\_ID, \_Duplication\_Index = fldObjectUID for newly-inserted row when \_Duplication\_Index is NULL (This line executes only when NEW record generated by the node, that is, a record that’s not being replicated).*

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;

1. *Updates \_Duplication\_Index = MIN(\_Duplication\_Index) in ALL Caravanas records that have the same “Identificadores\_str” value and with “FechaHora Registro” >= MIN(FechaHora Registro).*

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)))));

1. *Updates “TimeStamp” field in \_sys\_Trigger\_Tables to flag the python logic that a new duplicate record has appeared in the Caravanas table.*

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;

The new Tag INSERT TRIGGER Logic should be:

1. UPDATE Terminal\_ID, \_Duplication\_Index for NEW.ROWID when \_Duplication\_Index is NULL (New record generated in node).
2. WHEN the “fldCommissioned” is 0 (or NULL) in ALL of the records with the given “Identifiers\_str” value, run the Earliest Record Logic (already implemented).
3. WHEN the “fldCommissioned” is 1 in any of the records for the given “Identifiers\_str” duplicates, set record \_Duplicate\_Index = \_Duplicate\_Index from record with fldComissioned = 1.
4. UPDATE “TimeStamp” = NEW.”FechaHora Registro” in \_sys\_Triggers\_Tables, for DB\_Table\_Name == ‘Caravanas” AND ROWID == Flag\_ROWID.

Triggers. 2nd round. – 18 June 2024

The INSERT Triggers for tables that must handle duplication had to perform 3 independent tasks:

1. Append the \_sys\_Terminal value in NEW, LOCAL records (those generated by the node).
2. Update the \_Duplication\_Index values for all records that share the same “Identifiers\_str” field.
3. Update \_sys\_Triggers\_Table row with the TimeStamp data of the insertion of the new record, to notify Python that there’s a new INSERTtion in the table that must be processed.

With the inclusion of the concept of Physically Attached (PHATT) to the system logic, triggers have need to re-design step 2 above, to account for the logic generated by the Physical Attachment of tags, which takes priority over the logic of simply picking the Earliest Duplicate Record.

With this in mind, the Tag triggers will be split into 2 triggers:

* One that performs 1, 2, 3 above when there is a record with PHATT set to True (a Physical Attachment of the Tag onto an object has been recorded).
* One that performs 1, 2, 3 above when Physical Attachment has NOT been recorded for the Tag.

The question now is whether to record \_Duplication\_Index for all INSERTS, in which case duplication must be detected by checking count(“Identifiers\_str”) > 1, or whether to leave \_Duplication\_Index = NULL when there is no duplication for the record. This 2nd option is more efficient from the Python stand point, as records with \_Duplication\_Index == None can be easily singled out and ignored.

However, it is more complicated for the Trigger implementation: Must count the occurrences of “Identificadores\_str” value and act only when count(“Identificadores\_str”) > 1. This is done with:

(SELECT COUNT("Identificadores\_str") FROM "Caravanas" WHERE Identificadores\_str == NEW.Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL)) > 1

Problem here: how to set the 1st Duplication Index -> USE IFNULL() ??

***Most efficient solution. First Attempt. -> DEPRECATED.***

*In order to keep Triggers as clean as possible and avoid even more complicated logic to determine \_Duplication\_Index in non-duplicates and duplicates, the best option is to always populate the \_Duplication\_Index field in Caravanas, Animales, etc. When no duplicates existe, \_Duplication\_Index = UID\_Objecto of the same record.*

*Then, Python will have to COUNT(“Identificadores\_str”), (count > 1) to determine if there are duplicates to process.*

The implementation of this solution leads to the following for Caravanas.

1. Trigger for inserts while no PHATT has been recorded.

CREATE TRIGGER IF NOT EXISTS "Trigger\_Caravanas\_INSERT\_NO\_PHATT" AFTER INSERT ON "Caravanas" FOR EACH ROW

WHEN (SELECT SUM("Physically Attached") FROM "Caravanas" WHERE Identificadores\_str == NEW.Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL)) == 0

# Enters the Trigger ONLY when SUM(“Physically Attached”) for all records with same “Identificadores\_str” = 0.

BEGIN

# Sets record’s Terminal\_ID to the node’s own Terminal\_ID value ONLY when the INSERTed Terminal\_ID is NULL.

UPDATE "Caravanas" SET Terminal\_ID = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1) WHERE "Caravanas".ROWID == NEW.ROWID AND Terminal\_ID IS NULL;

# SETS \_Duplication\_Index to the \_Duplication\_Index value present in the record with MIN(“FechaHora Registro”) ONLY IF MORE THAN 1 Record with same “Identificadores\_str” are present in Caravanas table.

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" FROM "Caravanas" WHERE Identificadores\_str == NEW.Identificadores\_str AND (SELECT COUNT("Identificadores\_str") FROM "Caravanas" WHERE Identificadores\_str == NEW.Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL)) > 1 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)))));

# If only 1 record with “Identificadores\_str” value is present in Caravanas (Single tag record, no duplicates), SETS \_Duplication\_Index to the record UID\_Objeto value. This is the “initial” \_Duplication\_Index value for every record in the table.

UPDATE "Caravanas" SET '\_Duplication\_Index' = (SELECT DISTINCT UID\_Objeto from "Caravanas" WHERE "Caravanas".ROWID == NEW.ROWID AND "Caravanas".\_Duplication\_Index IS NULL);

# Finally, UPDATEs the TimeStamp value in \_sys\_Triggers\_Table to signal the Phyton logic that there are Duplicates to process in the Caravanas Table ONLY when the count of “Identificadores\_str” > 1. (1 means no duplicates, hence nothing to process).

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 (SELECT COUNT("Identificadores\_str") FROM "Caravanas" WHERE Identificadores\_str == NEW.Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL)) > 1;

END;

1. Trigger for inserts when and AFTER a PHATT operation has been performed on the Tag. The logic in red above is repeated here, with the difference that the \_Duplication\_Index value set is that of the Latest record with “Physically Attached” flag set to 1. There is ONLY 1 record with PHATT set to 1 in normal operation. The logic uses the last item only to cover from potential input errors that enter multiple PHATT operations for the same tag.

CREATE TRIGGER IF NOT EXISTS "Trigger\_Caravanas\_INSERT\_PHATT" AFTER INSERT ON "Caravanas" FOR EACH ROW

WHEN (SELECT SUM("Physically Attached") FROM "Caravanas" WHERE Identificadores\_str == NEW.Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL)) > 0

BEGIN

UPDATE "Caravanas" SET Terminal\_ID = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1) WHERE "Caravanas".ROWID == NEW.ROWID AND Terminal\_ID IS NULL;

UPDATE "Caravanas" SET '\_Duplication\_Index' = (SELECT DISTINCT \_Duplication\_Index FROM "Caravanas" WHERE Identificadores\_str == NEW. Identificadores\_str AND "FechaHora Registro" == (SELECT MAX("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 "Physically Attached" > 0)) AND ("Salida YN" == 0 OR "Salida YN" IS NULL) AND "Physically Attached" > 0)

WHERE "Caravanas".ROWID IN (SELECT "ID\_Caravana" FROM (SELECT DISTINCT "ID\_Caravana" FROM "Caravanas" WHERE Identificadores\_str == NEW.Identificadores\_str AND (SELECT COUNT("Identificadores\_str") FROM "Caravanas" WHERE Identificadores\_str == NEW.Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL)) > 1 AND ("Salida YN" == 0 OR "Salida YN" IS NULL)));

UPDATE "Caravanas" SET '\_Duplication\_Index' = (SELECT DISTINCT UID\_Objeto from "Caravanas" WHERE "Caravanas".ROWID == NEW.ROWID AND "Caravanas".\_Duplication\_Index 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 (SELECT COUNT("Identificadores\_str") FROM "Caravanas" WHERE Identificadores\_str == NEW.Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL)) > 1 ;

END;

***Most efficient solution. Second Attempt.***

Here, \_Duplication\_Index is left blank (NULL) when there are no duplicates (COUNT(“Identificadores\_str”) = 1) and the **trigger is only fired when COUNT(“Identificadores\_str”) > 1**.

The resulting triggers are more efficient and the Python logic is greatly simplified and more efficient too.

The 1st change is that Terminal\_ID is not set by Triggers anymore. It is established as a Default Value in all tables carrying it (Animales, Dispositivos, Geo Entidades, Caravanas) as the ID of the local node, set in table \_sys\_terminal\_id. This simplifies the requirements for Triggers.

Another change is that the update of \_sys\_Trigger\_Tables is done using Trigger\_Name, as follows:

*UPDATE \_sys\_Trigger\_Tables SET TimeStamp = NEW."FechaHora Registro" WHERE Trigger\_Name == "Trigger\_Caravanas\_INSERT\_NO\_PHATT" AND \_sys\_Trigger\_Tables.ROWID == \_sys\_Trigger\_Tables.Flag\_ROWID ;*

A \_Duplicate\_Index is found based on the type of logic (PHATT, NON-PHATT). If it is non-NULL, it is assigned to the duplicate record \_Duplication\_Index. If \_Duplicate\_Index is NULL, the UID\_Objeto belonging to the same record that yielded a NULL \_Duplication\_Index is assigned. Uses IFNULL() function for this.

2 Triggers are defined for Caravanas INSERT operations. **They both run on every INSERT**. The key factor is that only one of them will complete its actions: When there’s no Physical Attachment actions, the non-PHATT trigger executes.

When 1 record appears with “Physical Attachment” > 0, the PHATT trigger executes and the non-PHATT never executes again. (It fires, but the WHEN clause at the beginning fails and the actions are not executed).

The Resulting Triggers are:

**1). PHATT (“Physical Attachment” field > 0 for any of the duplicate records).**

*CREATE TRIGGER IF NOT EXISTS "Trigger\_Caravanas\_INSERT\_PHATT" AFTER INSERT ON "Caravanas" FOR EACH ROW*

*WHEN (SELECT SUM("Physically Attached") FROM "Caravanas" WHERE Identificadores\_str == NEW.Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL)) > 0 AND (SELECT COUNT("Identificadores\_str") FROM "Caravanas" WHERE Identificadores\_str == NEW.Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL)) > 1*

*BEGIN*

*UPDATE "Caravanas" SET '\_Duplication\_Index' = IFNULL((SELECT DISTINCT \_Duplication\_Index FROM "Caravanas" WHERE Identificadores\_str == NEW. Identificadores\_str AND "FechaHora Registro" == (SELECT MAX("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 "Physically Attached" > 0)) AND ("Salida YN" == 0 OR "Salida YN" IS NULL) AND "Physically Attached" > 0),*

*(SELECT DISTINCT UID\_Objeto FROM "Caravanas" WHERE Identificadores\_str == NEW. Identificadores\_str AND "FechaHora Registro" == (SELECT MAX("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 "Physically Attached" > 0)) AND ("Salida YN" == 0 OR "Salida YN" IS NULL) AND "Physically Attached" > 0))*

*WHERE "Caravanas".ROWID IN (SELECT "ID\_Caravana" FROM (SELECT DISTINCT "ID\_Caravana" 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 Trigger\_Name == "Trigger\_Caravanas\_INSERT\_PHATT" AND \_sys\_Trigger\_Tables.ROWID == \_sys\_Trigger\_Tables.Flag\_ROWID;*

*END;*

**2.) NON-PHATT (“Physical Attachment” field is 0 for ALL of the duplicate records).**

*CREATE TRIGGER IF NOT EXISTS "Trigger\_Caravanas\_INSERT\_NO\_PHATT" AFTER INSERT ON "Caravanas" FOR EACH ROW*

*WHEN (SELECT SUM("Physically Attached") FROM "Caravanas" WHERE Identificadores\_str == NEW.Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL)) == 0 AND (SELECT COUNT("Identificadores\_str") FROM "Caravanas" WHERE Identificadores\_str == NEW.Identificadores\_str AND ("Salida YN" == 0 OR "Salida YN" IS NULL)) > 1*

*BEGIN*

*UPDATE "Caravanas" SET '\_Duplication\_Index' = IFNULL((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)),*

*(SELECT DISTINCT UID\_Objeto 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" 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 Trigger\_Name == "Trigger\_Caravanas\_INSERT\_NO\_PHATT" AND \_sys\_Trigger\_Tables.ROWID == \_sys\_Trigger\_Tables.Flag\_ROWID ;*

*END;*

**Tag Inventory count logic, with the arrival of PHATT logic.**

With the implementation of PHATT logic, Inventory count is left as the last option for the assignment of \_Duplication\_Index: Only when PHATT bit is not set for any of de \_Duplication\_Indices of the chain will the code resort to counting Inventory activities for each of the UIDs.

Given that in the normal course of things the Physical Attachment of the Tag always comes first, it is expected that the use of Inventory counts to derive \_Duplication\_Index will occur rarely.

A third trigger is defined for Caravanas table:

3 - The trigger below sets the JSON field UID\_Objeto to the list of Tag uids associated with the Inventory Activity recorded. For normal object functions, the list is limited to 1 item: the uid for which the Inventory is recorded. For block execution, a list of all uids tied to this one activity is passed. The Trigger executes ONLY when the executing node is the local node. Does not execute for replicated records. And executes ONLY for TagInventoryActivity.

*CREATE TRIGGER IF NOT EXISTS "Trigger\_Data Caravanas Inventario\_INSERT" AFTER INSERT ON "Data Caravanas Inventario" FOR EACH ROW WHEN NEW.Terminal\_ID IS NULL*

*BEGIN*

*UPDATE"Data Caravanas Inventario" SET "Terminal\_ID" = (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1);*

*UPDATE "Data Caravanas Inventario" SET "UID\_Objeto" = json\_group\_array( "ID\_Caravana") FROM (SELECT "ID\_Caravana" FROM "Link Caravanas Actividades" WHERE "ID\_Actividad" == NEW."ID\_Actividad") WHERE "Data Caravanas Inventario"."ID\_Data Inventario" == NEW."ID\_Data Inventario";*

*END;*

**ANIMAL Triggers. 20-Jun-24.**

Animals are assigned Tag UIDs as their identifiers. These Tags have in turn their Identifiers resolved via the Caravanas Triggers.

When a Tag UID is assigned as an Identifier for an Animal, it is appended to the Animal record’s “Identificadores” field.

Animal Triggers then group duplicate Animal records by searching the “Identificadores” column and flag as a duplicate record all those records that share 1 or more identifier tags.

The logic for \_Duplication\_Index is pulling the Animal record with the earliest TimeStamp (Earliest Duplicate Record) for a given set of records with common “Identificadores” values: \_Duplication\_Index is dynamically updated to the UID\_Objecto of the record with the earliest TimeStamp value for a given set of records.

This logic makes sense in light of the adoption of the “Physical Attachment” concept that drives the logic for Tags duplicates given that, in principle, the earliest record created for an Animal is the action of assigning a Tag to the Animal: no other Animal record should have an earlier date than the record by which an Identifier is assigned to the Animal.

With this, the logic of pulling the Earliest Duplicate Record will eventually converge to the record corresponding to the tag assignment to the Animal, and while it does not converge, the logic of linking all duplicate records via the \_Duplication\_Index will render the Earliest Duplicate Record available at all times.

The resulting trigger for Animales INSERT is:

*CREATE TRIGGER IF NOT EXISTS "Trigger\_Animales\_INSERT" AFTER INSERT ON "Animales" FOR EACH ROW*

*BEGIN*

*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)))));*

*UPDATE \_sys\_Trigger\_Tables SET TimeStamp = NEW."FechaHora Registro" WHERE Trigger\_Name == "Trigger\_Animales\_INSERT" AND \_sys\_Trigger\_Tables.ROWID == \_sys\_Trigger\_Tables.Flag\_ROWID;*

*END;*

Side note: The update of the \_Duplication\_Index value based on the count of repeat (duplicate) identifiers recently implemented in Caravanas Triggers was already working in the Animales INSERT trigger with the use of the INSTR() call:

*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))))*

In the code snippet above the value of \_Duplication\_Index is only updated when *INSTR(NEW.Identificadores\_str, json\_each.value) > 0,* that is, when there is as a minimum 1 Identifier in the “Identificadores” JSON Lists (for all the records in the table) that matches the identifiers passed in *NEW.Identificadores\_str*. If no repeat identifiers are found the condition *INSTR(NEW.Identificadores\_str, json\_each.value) > 0* is not met and the result of SELECT is NULL, hence setting \_Duplication\_Index = NULL when there are no duplicates.

***This Animales Trigger, as is, should be fully portable to Dispositivos and Personas, provided the “Identificadores” field is defined as a JSON List.***

sdf

**A digression on json\_each, json\_tree functions and the need for their use in some triggers.**

The main use found for them is in "Trigger\_Animales Registro De Actividades\_INSERT".

There, each INSERT in the table must append the type of animal (animalClassID) to the field “Updated\_By”.

If appending is done blindly, the field will rapidly grow with multiple repeat values, so the logic must discard repeat values and keep only 1 value for each animalClassID. With that working, the “Updated\_By” field would look like [1,2,3,4] when updates by animal classes 1, 2, 3, 4 have been entered in the db.

In order to ignore repeat values, the “Updated\_By” field is CAST into a string and a INSTR search is performed with NEW."ID\_Clase De Animal", as follows:

*INSTR(CAST(json\_each.json AS TEXT), NEW."ID\_Clase De Animal") == 0*

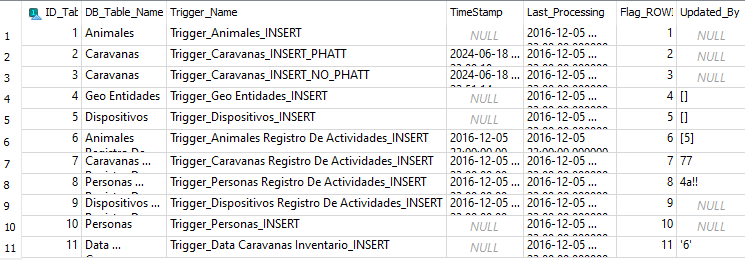
The main use of the json\_each(), json\_tree() functions come here, to generate a table from the “Updated\_By” column and iterate through its values with json\_each.json in order to check the condition above.

json\_each() and json\_tree() return database tables. This is very important to understand. Then, those tables are treated in the same way as any other db table.

See: <https://sqlite.org/json1.html#jins>, <https://database.guide/sqlite-json_each/>

Example.

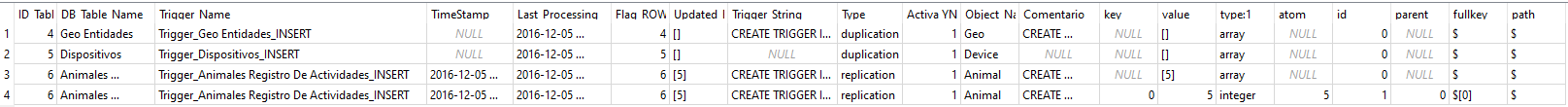
Original \_sys\_Triggers\_Table:



Command with json\_tree():

*SELECT \* FROM \_sys\_Trigger\_Tables, json\_tree(\_sys\_Trigger\_Tables."Updated\_By")*

Resulting json\_tree table:



Note that ID\_Table #6 is repeated as json\_tree() first creates a row for the JSON list in that row ([5]) and then iterates through the elements and creates another row for the value 5.

json\_tree() lists empty json arrays (rows 1, 2).

Command with json\_tree():

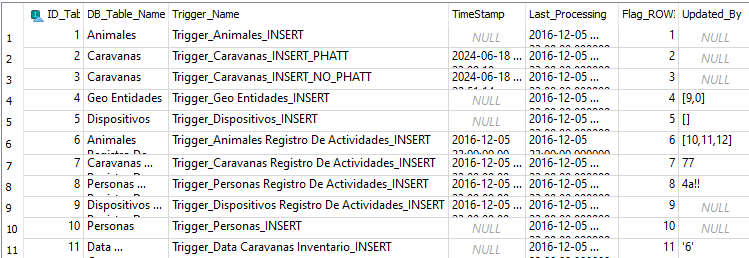
*SELECT \* FROM \_sys\_Trigger\_Tables, json\_each(\_sys\_Trigger\_Tables."Updated\_By")*

Resulting json\_each() table:

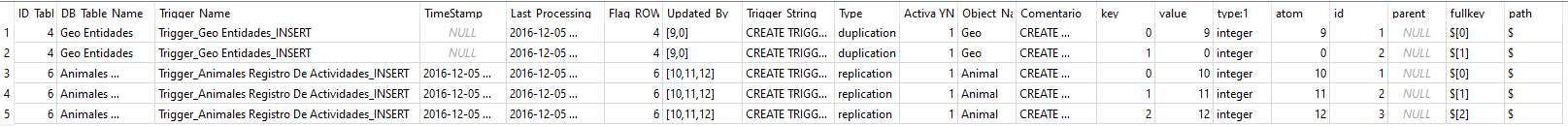


Very important: json\_each() does NOT pull empty json arrays. This is critical, as [] has to be handled by the trigger when a list becomes empty (nothing pending processing).

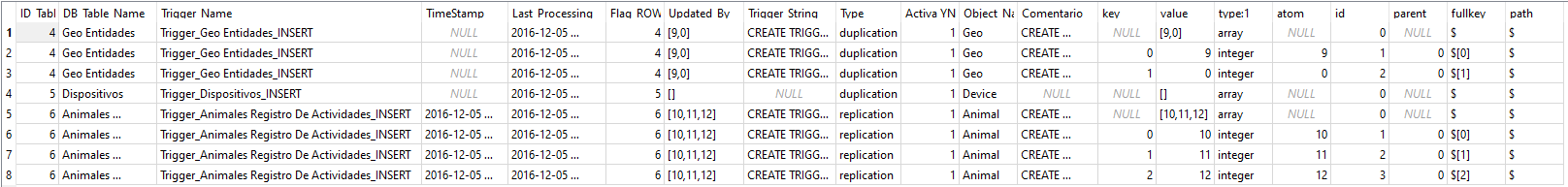
Another table:



json\_each():



json\_tree():



Here, the empty json array shows.

The method of CASTing to string is flawed as it will yield incorrect results comparing single numbers: 1 will match 1 but will also match 10, 11, 31, 2001.

So another comparison method must be implemented.

json\_extract() returns string representations of arrays, so it won’t do either.

Using json\_each() gives the final Trigger:

*CREATE TRIGGER IF NOT EXISTS "Trigger\_Animales Registro De Actividades\_INSERT" AFTER INSERT ON "Animales Registro De Actividades" FOR EACH ROW WHEN NEW.Terminal\_ID IS NOT NULL AND NEW.Terminal\_ID != (SELECT Terminal\_ID FROM \_sys\_terminal\_id LIMIT 1)*

*BEGIN*

*UPDATE \_sys\_Trigger\_Tables SET TimeStamp = NEW."FechaHora Registro" WHERE Trigger\_Name == "Trigger\_Animales Registro De Actividades\_INSERT" AND \_sys\_Trigger\_Tables.ROWID == \_sys\_Trigger\_Tables.Flag\_ROWID;*

*UPDATE \_sys\_Trigger\_Tables SET "Updated\_By" = json\_set(IFNULL((SELECT "Updated\_By" FROM \_sys\_Trigger\_Tables WHERE Trigger\_Name == "Trigger\_Animales Registro De Actividades\_INSERT" AND \_sys\_Trigger\_Tables.ROWID == \_sys\_Trigger\_Tables.Flag\_ROWID*

*AND json\_valid((SELECT "Updated\_By" FROM \_sys\_Trigger\_Tables WHERE Trigger\_Name == "Trigger\_Animales Registro De Actividades\_INSERT" AND \_sys\_Trigger\_Tables.ROWID == \_sys\_Trigger\_Tables.Flag\_ROWID))*

*AND json\_type((SELECT "Updated\_By" FROM \_sys\_Trigger\_Tables WHERE Trigger\_Name == "Trigger\_Animales Registro De Actividades\_INSERT" AND \_sys\_Trigger\_Tables.ROWID == \_sys\_Trigger\_Tables.Flag\_ROWID)) == 'array'), '[]'), "$[#]", NEW."ID\_Clase De Animal")*

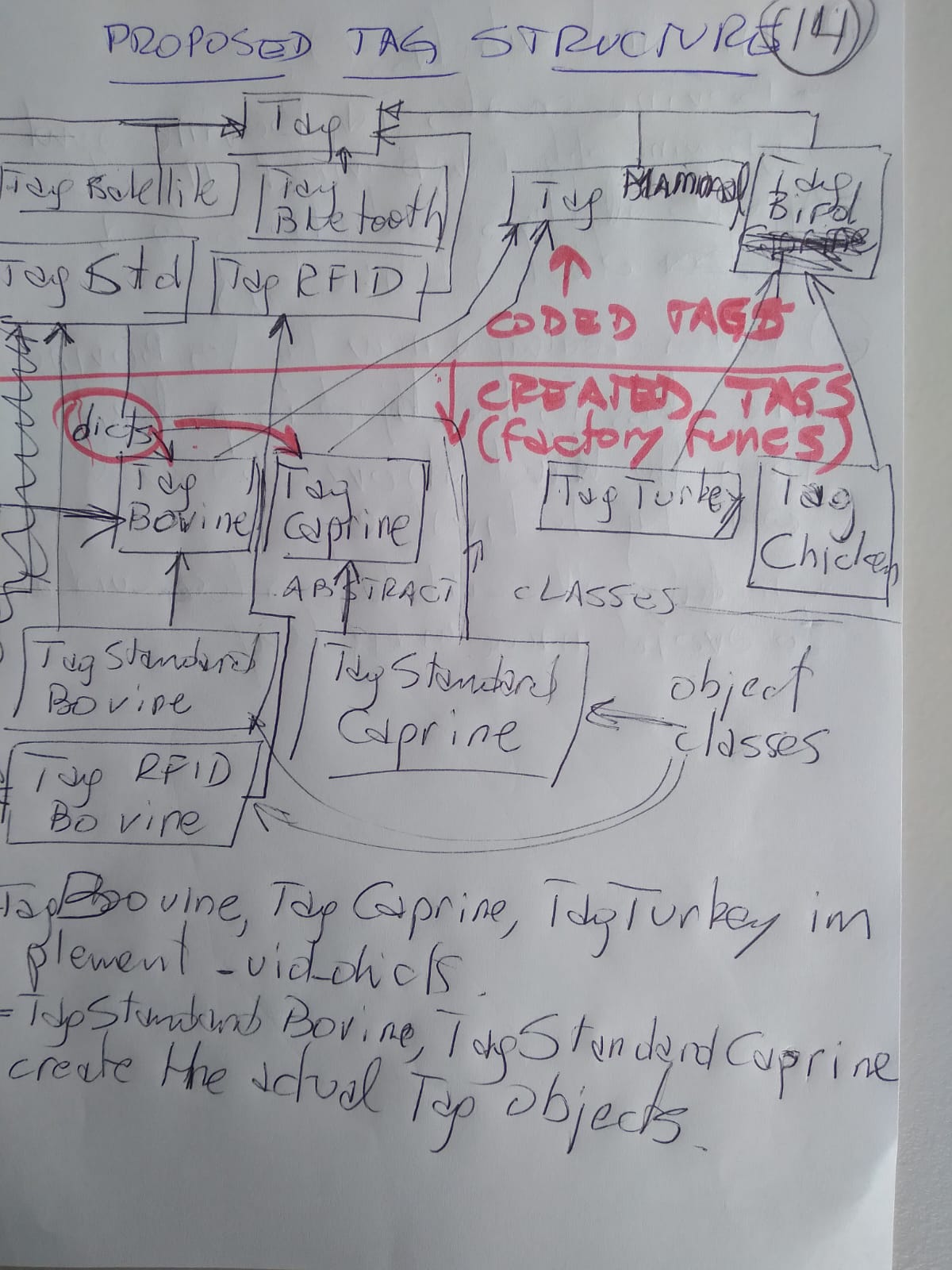
*WHERE Trigger\_Name == "Trigger\_Animales Registro De Actividades\_INSERT" AND \_sys\_Trigger\_Tables.ROWID == \_sys\_Trigger\_Tables.Flag\_ROWID*

*AND NEW."ID\_Clase De Animal" NOT IN (SELECT json\_each.value FROM \_sys\_Trigger\_Tables, json\_each(\_sys\_Trigger\_Tables."Updated\_By") WHERE \_sys\_Trigger\_Tables.ROWID == \_sys\_Trigger\_Tables.Flag\_ROWID AND Trigger\_Name == "Trigger\_Animales Registro De Actividades\_INSERT" AND json\_valid(json\_each.json));*

*END;*

**New Tag Structure.**

Tag structure is changed to support multiple inheritance from the Object side (Bovine, Caprine, Chicken, Device, etc) and from the Technology side (TagStandard, TagRFID, etc). As follows:



The design logic is:

* Tag technologies will evolve and new technologies will be added with time.
* Each tag technology will require a support management structure equivalent to a Device Driver, with specific attributes and methods to leverage the technology processes and data management.
* On the other side of the fence, Tag classes will be defined for each Animal and Device class: These classes will be abstract and will implement the \_active\_uids, \_duplicate\_uids\_dict for each object class, in order to give compartmentalization of Tag objects based on Object class.

To this end, base classes will be coded in modules: TagMammal, TagBird, TagMachinery, TagVehicle on the object side,

Technology classes TagStandard, TagSatellite, TagRFID on the tag technology side.

Below these classes, factory function \_create\_subclass() will 2 types of classes:

**Parent classes (abstract class):** TagBovine, TagCaprine, TagVehicle, TagTurkey, etc

**Child classes (object classes):** TagStandardBovine(TagBovine, TagStandard), TagRFIDBovine(TagBovine, TagRFID).

These classes create the actual Tag objects during system initialization and via tag creator functions (assignTag, etc).

The Child classes inherit from Technology Classes and Parent classes*, in that order* (see why below), so as to have access to all the attributes and methods from the technology side and from the Object side.

With this structure:

* Child (Object classes) will always have a Technology class and an Object class (Bovine, Caprine, Machine, etc) as parents.
* Creation of activities: As Activity Objects for each Object class are created in EntityObject.\_\_init\_subclass\_\_() function -> an ad-hoc “mro” must be considered to resolve for the cases where a method or attribute has the same name in both parent classes (Ex.: both TagStandard and TagBovine implement a certain method with the same name.

The solution (for now) is given by the order in which the Activity objects for both parent classes are created: The Technology Class Activity creator function is executed last, so for any common names, the attribute in the Object class (created via setattr func) will be overwritten with the value from the Technology class and this will be the prevailing attribute.

*This is very primitive and, in principle, Technology classes and Object classes (Tag class) should have NO common names. Any cases of common attribute names must be considered carefully.*

To get there:

1. Implement \_active\_uids\_dict, \_\_active\_duplication\_index\_dict, \_identifiers\_dict **in Parent Classes** TagBovine, Tag Caprine, etc. (as it is now). **Rename** \_active\_uids\_dict, \_\_active\_duplication\_index\_dict, \_identifiers\_dict (mangled name) inside \_create\_subclass() function to private names (\_active\_uids, etc.) to be able to access the dictionaries from the child classes (which are the ones implementing the actual Tag objects).
2. Implement a tagTechDict in the Parent classes of the form {tagTechnology: <Child class>, }. This dict populated by the \_create\_subclass() function, in a for loop that traverses all the Technology classes and creates 1 Child class for each Technology.

It is to be accessed by the Tag creating and initializing functions to pull the proper Child (object) class based on object type and technology type, and create the Tag object of the right class.

1. The child classes (TagStandardBovine, TagRFIDBovine, TagStandardTurkey, TagSatelliteVehicle, etc.) are created by the \_create\_tag\_subclass() function, are added to the tagTechDict dictionary (as described above) and are encapsulated there. Only accessed via the dictionary.

The resulting Tag class creator function is below:

@classmethod  
def \_create\_subclass(cls, cls\_name, \*, obj\_class=None, tag\_tech\_class=None):  
 *""" Class Factory function.  
 Creates Tag subclasses for every Animal type that defines the \_\_uses\_tag attribute.  
 Classes created implement the dictionaries and data structures below and use all the methods from class Tag.* ***@param*** *cls\_name: str. Class Name given to the class being created ('TagBovine', 'TagCaprine', etc).* ***@param*** *obj\_class: Animal, Device subclass that will use this Tag class (class Bovine, class Caprine, etc).  
 """* def init\_parent(self, \*args, \*\*kwargs): # Can't remove self from here!! It must always be 1st arg in \_\_init\_\_()  
 super().\_\_init\_\_(\*args, \*\*kwargs)  
  
 # Returns Tag sub\_class template. Base class is cls. cls\_name=TagBovine, TagCaprine, TagChicken, TagDove, etc.  
 parent\_class = type(cls\_name, (cls, ), {  
 '\_objectClass': obj\_class, # <class Bovine>, <class Caprine>, etc.  
 '\_active\_uids\_dict': {},  
 # {fldObjectUID: fld\_Duplication\_Index} --> fld\_Duplication\_Index IS an object UID.  
 '\_duplic\_index\_checksum': 0,  
 # sum of \_active\_uids\_dict.values() to get changes and update \_active\_uids\_dict  
 '\_active\_duplication\_index\_dict': {}, # {fld\_Duplication\_Index: (fldObjectUID, dupl\_uid1, dupl\_uid2, ), }  
 '\_identifiers\_dict': {},  
 # {fldObjectUID: fldIdentificadores} --> dict to access Tag identifiers without reading from DB  
  
 '\_tagObjectsClasses': {}, # {tagTech(str): <Child (object) class>, } ex: {TagStandard: <TagStandardBovine>}  
 '\_\_init\_\_': init\_parent, # \_\_init\_\_() to pass up the chain.  
 })  
  
 def init\_child(self, \*args, \*\*kwargs):  
 super().\_\_init\_\_(\*args, \*\*kwargs)  
  
 # Creation of classes that will instantiate actual Tag objects.  
 for name, c in cls.getTagTechClasses().items():  
 child\_name = name + parent\_class.\_\_name\_\_ # TagStandardBovine, TagRFIDMachine, tagBluetoothChicken, etc.  
 child\_class = type(child\_name, (c, parent\_class), {  
 # IMPORTANT: The order of parent classes determines the override mechanism for attributes with same name.  
 '\_\_init\_\_': init\_child, # \_\_init\_\_() to pass up the chain  
 })  
 parent\_class.\_tagObjectsClasses[name] = child\_class # {tagTech: TagObjectClass (TagStandardBovine, etc)., }  
 return parent\_class

1. During initialization of objects classes (Bovine, Caprine, Machine, etc.) the proper Tag object classes must be assigned to the ‘\_myTagClass’ attribute in the Object classes.
2. The tag creation functions (initializeTags(), assignTags) uses the’ \_myTagClass’ attribute value to invoke the proper object constructor. Now, the tag creation must be done on a **‘per tag’ basis**, that is, the technology type and animal type must be pulled for each tag in order to invoke the proper Tag constructor.

With this change, the \_myTagClass will remains unchanged (returning TagBovine, TagCaprine, TagMachine, etc.)

The difference is that the getObject() function must be redefined using the tag technology data to pull the right class from the getTagTechClasses dict and return an object of that class.

All in all, the changes to derive the proper class and return objects of that class will must be implemented in Tag.getObject()which pulls uids from tblCaravanas and based on the technology field pulled from db and the cls caller function must return the proper Tag Class. And also in and AltaActivity.\_\_alta() function.

Sdfsdf krnl\_abstract\_class\_activity.py.\_paCreateExecInstance:2323

**25-Aug-2024: Tag Entry**

The Tag Input process is outlined here. Some ideas.  
The TagInput Activity that must handle all tag inputs for the system. A tag Input is defined as an action  
whereby a tag object is scan or read, hence an inventory on it must be performed.

Tag Inventories are then to be handled entirely by the tag activities and NOT cascaded or delegated to any other object class. ***Enforce and strive for encapsulation. It usually is the best possible answer.***

Tag entries are thought of in 3 ways at the time:

1. **Keyboard Input**: Entry is done manually within the front-end application via Terminal keyboard.
2. **Terminal Scan**: tag is scanned by the front-end application using a proper scanning application.
3. **Linked Device Input**: Data from a Tag scan is sent to the Terminal via Bluetooth, NFC or WIFI. It must be processed by the application.

For this, a proper device link protocol must de devised for the front end so scans are properly processed by the terminal. Think of a situation with multiple scanning devices and multiple Terminals interacting.

This will probably be resolved by a dedicated thread that waits for scan data from linked and validated scanning devices.

1. File Input: Scanning devices generate a csv file that is processed by the front-end application when received.

In every of these 4 cases, the actions performed involve a physical readout of the tag, hence an inventory will be recorded for each successfully scanned tag.

*Must strive for this to be pretty much the only Inventory Activity for tags, as all activities involved tagged objects that require the object being identified (via its tags) will forcefully involve a tag scan, hence and inventory. See how to implement this as much and as far as possible.*

Sdfsdfsdf

Sdfsd

Pandas.

Pandas DataFrame is the standard for database record handling across the system.

DataFrame objects are associated to a given table name using a database accessor: “db”.

To implement the association of DataFrames to db tables, the following is to be implemented:

* A DataBaseTableAccessor class that defines attributes and methods.
* Attributes defined in the Accessor class are:
  + \_tbl\_name (str)
  + \_field\_names (dict)
  + \_reverse\_names (dict)
* Panda sql reading functions pd.read\_sql\_query(), read\_sql\_table() are decorated with a dedicated wrapper that:
  + Converts all column names to key field names.
  + Initializes the 3 attributes above.
  + Converts all Timestamp columns to datetime data.
* An accessor create() method that returns a pandas dataframe with all the accessor attributes set.
* setrecords() function: accepts a db dataframe as an argument (must have its db Accessor attributes set) and writes it to db.
* setRecord() function: accepts a table name and a dictionary as arguments and writes 1 row to db.
* getrecords() function: accepts a table name, a list of field names and a “WHERE” clause (string) and rerturns a pandas dataframe.

Sdfsd

**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 a specific object. The index of the dataframe is the object’s uid.

For this, get\_mem\_data(), set\_mem\_data() will be modified to operate with DataFrame objects.

**Chapter 13: Serializer Locks.**

Concurrency management class that internally uses a Condition object to serialize access to shared resources.

The logic is that the 1st thread to access the resource never waits and goes straight to handle the resource.

If, while the 1st thread is working in the protected code block, other threads attempt to access, they wait in line until notified by the internal Condition to proceed.

There is a special behavior designed in the class that keeps track of the count of the threads accessing the protected block and enables all threads, by setting, to NOT block and continue operation until a blocking/waiting condition is specified in the code.

With this behavior, the program can acquire() the SerializerLock object in a non-blocking manner when one wants to start serializing access (this is the only moment when one can keep track of the order of access to the shared resource). For instance, read some data from db that must be processed and then assigned to a memory resource.

After this, the order of execution of the threads is undetermined (only defined by the OS internal scheduler). With the internal counter, the user can opt for 2 execution stances:

* Can execute the protected code for the 1st thread that accessed the shared resource (that read db).
* Can execute the protected code for the LAST thread that accessed the shared resource. T

This option is used when accessing \_init\_uid\_dicts structure by front or by background threads, as shown below:

# TODO(cmt): all the code below down to the end is protected from concurrenty re-entry by other threads.  
# All subsequent calls to \_init\_uid\_dicts() will go to wait() until release() runs its notify() line.  
with cls.\_serializer\_lock(blocking=False) as access\_count:  
# access\_count = cls.\_serializer\_lock.acquire(wait=False)  
 # Starts serializing access here, by detecting all concurrent calls to \_init\_uid\_dicts and icrementing  
 # access\_count for each instance of \_init\_uid\_dicts that accesses the database. It is needed in order to  
 # identify later on the last one to access database. This will be the instance that updates \_active\_uids\_df.  
 read = pd.read\_sql\_query(cls.\_sql\_uids(), SQLiteQuery().conn, chunksize=cls.\_chunk\_size)  
  
 # IMPORTANT: tee() is NOT thread-safe. However, read and ittr are local vars. They won't be accessed by  
 # other threads. With them being local, we can use wait=False in \_serializer\_lock above.  
 ittr = it.tee(read, 3) # 2 iterators used for processing; 1 (ittr[2]) is assigned to \_active\_uids\_df.  
  
 for df in ittr[0]: # Detects empty frames.  
 if not df.empty:  
 break  
 # Exits with error if 1st dataframe in iterator is empty.  
 val = f"ERR\_DBAccess cannot read from table {cls.tblObjDBName()} internal uid\_dicts " \  
 f"not initialized. System cannot operate."  
 krnl\_logger.warning(val)  
 # cls.\_serializer\_lock.release() # TODO(cmt): Line not required when using context manager.  
 raise sqlite3.DatabaseError(val)  
  
 dupseries = None  
 for df in ittr[1]:  
 # Sets up/updates a Series with duplicates for ease of access to duplicates.  
 if any(df['fld\_Duplication\_Index'].notnull()):  
 not\_nulls = df.loc[df['fld\_Duplication\_Index'].notnull()]  
 temp\_dupl = not\_nulls.groupby('fld\_Duplication\_Index')['fldObjectUID'].agg(set)  
 if dupseries is None:  
 dupseries = temp\_dupl  
 else:  
 dupseries.append(temp\_dupl, ignore\_index=True)  
  
 # The 'if' below:  
 # access\_count == cls.\_serializer\_lock.total\_count -> Enters 'if' for LAST execution instance of func.  
 # access\_count == 1 -> Enters 'if' for the FIRST execution instance of func. To be used where needed.  
 if access\_count == cls.\_serializer\_lock.total\_count:  
 cls.\_serializer\_lock.\_wait() # Enters wait, until notified. To serialize access to \_active\_uids\_df.  
 with cls.\_slock\_obj\_dataframe: # cls lock for memory resources. Used by all funcs that access \_active\_uids\_df.  
 cls.\_active\_uids\_df = ittr[2] # SHARED resource. Once notified proceeds with the writes to mem.  
 if isinstance(dupseries, pd.Series):  
 try:  
 pd.testing.assert\_series\_equal(dupseries, cls.\_dupl\_series)  
 except (TypeError, AssertionError, ValueError):  
 cls.\_dupl\_series = dupseries.copy() # SHARED resource. Accessed from background threads.

This is a substantial tool to manage concurrent access to a resource by multiple threads. A key feature is that multiple objects can be defined, specific to each resource to be shared, and the access to such resource will in no way block/interfere with the rest of the code executing in the system.

As an example, the Activity Classes that implement Memory Data, each use a dedicated SerializerLock object (defined as a Class attribute) to manage concurrent access to each Activity’s Memory DataFrame.

06Aug24: **A new flag, *read\_only,* will be added as an experimental feature.**

This flag will make the thread using it “invisible” to the serializer lock logic. This is implemented by NOT updating the access\_count and total\_count internal counters.

The intended behavior when a thread sets this flag to True is:

1. The thread will enter a wait state once it acquires the lock, if there are any other threads using the lock.
2. The thread will NOT block other threads from accessing the thread resource (thus the “invisibility” of the thread). This feature also must preclude the threads from making write-access to the shared resource, effectively enabling it for read-only operations.

Not sure how this could be used in real life, but the concept is this, to be tested.