Tables naming convention:

<**tableType**>_<**contextIdentifier**>_<**nodeIdentifier**>_<**UUID**>
ex.

table_regression123_node3_fe7a872ac7
view_regression123_node1_aa987ce212
merge_regression123_global_fe7a872ac7

tableType:
possible values: ***table***, ***view*** or ***merge table***.
Concerning ***remote tables***, they always have the name of the "native" table, a different name cannot be set.
The tableType is solely for ***human readability***

contextIdentifier:
For now the only context we are concerned with is that of an **algorithm**, so this will consist of the name of the algorithm and a number.
The contextIdentifier will be used to allow for discrimination of tables relevant to the execution context and will be ***used*** *by the* ***code*** *(see below)*
The contextIdentifier will be produced by the controller

nodeIdentifier:
possible values: ***global*** or ***node<1-N>***
Defines a node identifier denoting where the table "natively" exists.
Having the node identifier in the table name allows knowing **in which node the actual table exists**, which is needed when a remote node needs to create a remote table.
The nodeIdentifier will be produced by the node

UUID (UniversallyUniqueIdentifier):
A hex(could also be a decimal..) number of size 10 (maybe needs to be longer, to ensure uniqueness??) that must always(well, with high probability) be **unique along all nodes**.
The UUID will be produced by the node

Here is how the celery task/functions need to be refactored:

The highlighted ones are the ones that implement the minimum functionality needed. The rest will be needed for debugging, so the developers outside the node services will not have to go look inside the node's runtime to figure out why the task they queued is not producing the expected behavior.
The gray highlighted ones are probably not needed after the `clean_up(context_id:str)` **is implemented**

TABLES:
naming convention: *table_<contextIdentifier>_<nodeIdentifier>_<UUID>*

- `get_tables(context_id:str)→ List[str] #all table names on this node with this context_id`
- `get_table_schema(table_name:str)→ List[ColumnInfo] #do not include table name in the response`
- `get_table_data(table_name:str)→ TableData`

  `Also change the order of attributes in TableInfo, 1st schema, 2nd data`
- `create_table(context_id:str, schema: List[ColumnInfo])→ str #table name`

  `return only the table name, not the schema`

- `delete_table(table_name: str) → some kind of success response?` **TBD**

NOTE: for all tasks, a comprehensive error response should be defined, for the cases where the task failed either because of bad input parameters or for any other reason

VIEWS:

naming convention: *view_<contextIdentifier>_<nodeIdentifier>_<UUID>*

- `get_views(context_id:str)→ List[str] #all view names on this node with this context_id`
- `get_view_schema(view_name:str)→ List[ColumnInfo] #do not include view name in the response`
- `get_view_data(view_name:str)→ TableData`
- `create_view(context_id:str,`**`???`**`)→ str` **TBD**

  We need to discuss the view creation more, I still do not fully understand how the data is organised. Pathologies, datasets etc.
- `delete_view(view_name: str) → some kind of success response?` **TBD**

REMOTE TABLES:

- `get_remote_tables(context_id:str)→ List[str] #all remote table names on this node with this context_id`
- `create_remote_table(table_name:str) → some kind of success response?` **TBD**
  No table name needs to be returned, remote table names are by default the same as the “native” table name
- `delete_remote_table(table_name:str)→ some kind of success response?` **TBD**

NOTE: **remote tables** will only be created **from local nodes to global node** and the **other way around**, never between local nodes. To that end, the global node needs to “know” the **addresses to all local** nodes’ monetdb servers and all local nodes need to “know” the **address of the global** node’s monetdb server. We need to come up with a **mechanism** for that functionality

MERGE TABLES:

naming convention: *merged_<contextIdentifier>_<nodeIdentifier>_<UUID>*

- `get_merge_tables(context_id:str)→ List[str] #all merge table names on this node with this context_id`

  (Merge tables will only be needed in the global node but this should occur naturally, we do not need a mechanism to enforce this)
- `create_merge_table(context_id:str, schema: List[ColumnInfo])→ str #table name`
- `update_merge_table(table_name: str,table_names:[str]):→ some kind of success response?` **TBD**

  table_names are a list of strings, the names of the tables to be merged
- `delete_merge_table(table_name:str)→ some kind of success response?` **TBD**

UDFs:

- `get_udfs() -> List[str]`

  returns the header of all the going-to-be udfs the node “knows” about, so it must return the header of the “pure” python syntax function, before it passes through the so-called udfGenerator

- `exec_udf(context_id:str, udf_name: str, input: List[Parameter]) -> str #table name`

  `List[Parameter]` is not very clear yet, needs more discussion **TBD**

- `get_generated_udf(udf_name: str, input: List[Parameter]) -> str`

  takes the same input parameters the exec_udf would take and returns the code generated by the so-called udfGenerator, without executing the udf

GENERAL:

- `clean_up(context_id:str)→` some kind of success response? **TBD**

  the controller will call this at the end of the context execution, so the node deletes all tables with this context_id

  The order of dropping the tables can become a bit complex as merge or remote tables must be dropped before their dependencies get dropped otherwise monetdb will complain and table cleaning will not be complete and can lead to “leaks”