MORIA User Manual

Austin Keller

April 4, 2025

Contents

Preface

This document is a comprehensive user manual for the **MO**ngodb **R**epository for **I**nformation and **A**rchiving (MORIA). Provides detailed instructions on how to use the database effectively, including access, setup, configuration, and troubleshooting.

1 Introduction

1.1 Overview

To fully realize the potential of high-repetition-rate (HRR) high-energy-density (HED) experiments and/or applications, all measurements from each subsystem (laser, target, diagnostics) need to be appropriately labelled and archived in real time (10 - 100 MB/s, 1 - 10 PB/yr). As advancements are made in ML/AI analysis routines, metrics derived from analyzed data will be used to provide feedback control to the laser and targetry subsystems, and aid in developing/improving models of the system performance.

1.2 Key Features

Given the shift toward a guided data organization solution for HRR HED experiments and applications, the MongoDB database management system was selected primarily due to several key attributes that make it well-suited to meet these requirements:

- **Horizontal scaling**: As data volumes increase (with the addition of diagnostics and increasingly long rep-rate experiments), the capacity of the database system can be increased by simply adding more computational nodes.
- **Integration with GridFS**: The GridFS file system allows storage of arbitrary, non-scalar data such as images along with associated metadata. MongoDB has created an API for reading such data directly from the database, eliminating the need to store files separately, thereby preventing issues with data integrity and consistency.
- **Open-source and well documented**: There are many officially supported libraries for accessing MongoDB databases in Python, Matlab, and C++. Full list at MongoDB Supported Languages. Extensive MongoDB documentation can be found here MongoDB Docs.
- **Dynamic**: Allows for seamless changes to the document schema, integration of additional devices, and other modifications without disrupting querying or archiving functionality.

1.3 System Requirements

Below is list of requirements necessary to utilize this MongoDB framework.

- **Hardware**: Dedicated server(s) running standard Linux operating systems.
- **Software**: Install MongoDB 8.0+ (MongoDB Installation Guide) and the MORIA API (GitHub) onto to your server(s).

2 Database Structure

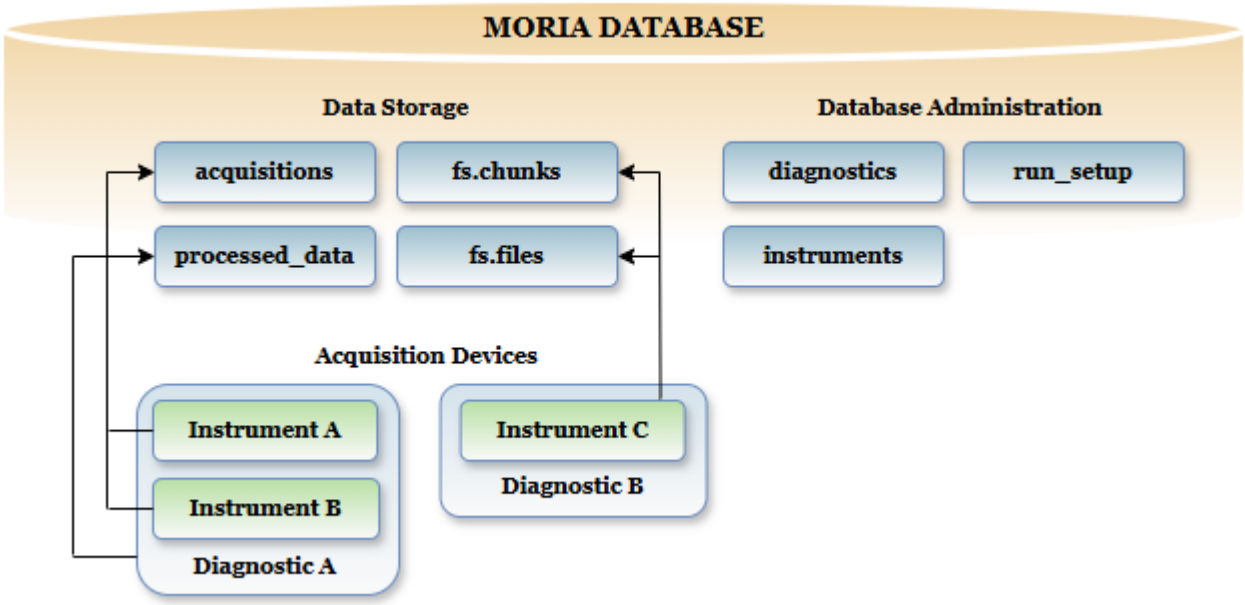


Figure 1: MORIA database schema. The MongoDB currently contains four collections used in data storage (`acquisitions`, `processed_data`, and `fs.files/fs.chunks`) and three responsible for administration (`diagnostics`, `instruments`, and `run_setup`). More information can be found in Tables and Fields.

MORIA is organized and generalized at the diagnostic level to lead the paradigm shift from shot-based to diagnostic-based data archiving. Multiple diagnostic collections can be queried at the same time, but only the diagnostic information that is of interest to the specific application need be opened at anyone time, allowing for more efficient use of computational resources.

To store data and metadata, MongoDB uses a generic collection structure.

- **Collection**: is a grouping of MongoDB documents, similar to a table in relational databases. Collections do not require a predefined schema, meaning that documents within the same collection can have different fields and structures.
- **Document**: is a set of key-value pairs and is similar to a row in a relational database table, but more flexible.

This type of structure is required because data acquired in HRR HED experiments can take multiple forms: singular numbers (e.g., trigger timing, shot number), 1D arrays (e.g., diode voltage traces), and images (e.g., CMOS or CCD images).

Raw data and associated metadata are stored in documents within the acquisitions collection (e.g., scalars, arrays) or within `fs.files/fs.chunks` for images or large data files. The large files are divided into smaller chunks using MongoDB's GridFS functionality to enhance archival and retrieval efficiency.

Additionally, the database includes several administrative collections. Two of these maintain lists of all `diagnostics` and `instruments`, while the `run_setup` collection is used to define global experimental parameters.

In the example shown in Figure 1, Instrument C archives image files, with the data stored in fs.chunks and the metadata in `fs.files`. In contrast, Instruments A and B, which record scalars or arrays and associated metadata, archive their data in `acquisitions`. As instruments are replaced or changed over time, as long as they are tagged with the same diagnostic label in their metadata, all data and associated metadata from the diagnostic can be queried without specifying the hardware.

The `processed_data` collection stores post-processed data, typically at the diagnostic level, as well as real-time experimental information. In this example, it contains the results of Diagnostic A.

2.1 Indexes and Constraints

The metadata fields **shot_number** and **device_name** form a single compound index. This index is applied exclusively to `fs.files/fs.chunks` and `acquisitions` and is designed to ensure uniqueness across individual devices for a single laser pulse.

3 Database Initialization

Instructions for initializing the database for the first time, following the configuration setup outlined in System Requirements.

1. Initialize the MongoDB service: As root user,
   - To start the service: `service mongod start`.
   - To stop the service: `service mongod stop`.
2. Clone or copy the online MORIA GitHub repository (moria) to a universally accessible directory on your server.
3. Navigate to `database_utils` within the new directory and run `create_database <database name>`. This command will setup a new database with the structure described in the schema overview. Once this is done you will be archive and query from the database.
4. Add the instruments and diagnostics unique to your system to the `instruments` and `diagnostics` collections respectively. This process is managed through the Administration API functions. To add entries, use `add_instrument` and `add_diagnostic`. To remove entries, use `remove_instrument` and `remove_diagnostic`.
   - The **instrument** tag is designed to categorize similar devices—regardless of manufacturer or model—under a unified classification. The **diagnostic** tag provides an additional layer of abstraction by grouping together the specific devices required to measure a particular output. This structure enables users to query data by diagnostic for a specific shot or run, retrieving all relevant raw and processed data without needing to reference individual hardware components.

     When adding an instrument, an additional parameter, `gridfs_bool`, must be specified. This parameter indicates whether the device data should be archived in GridFS. It must be set to **True** if the data may exceed **16 MB** (such as camera images or large arrays).

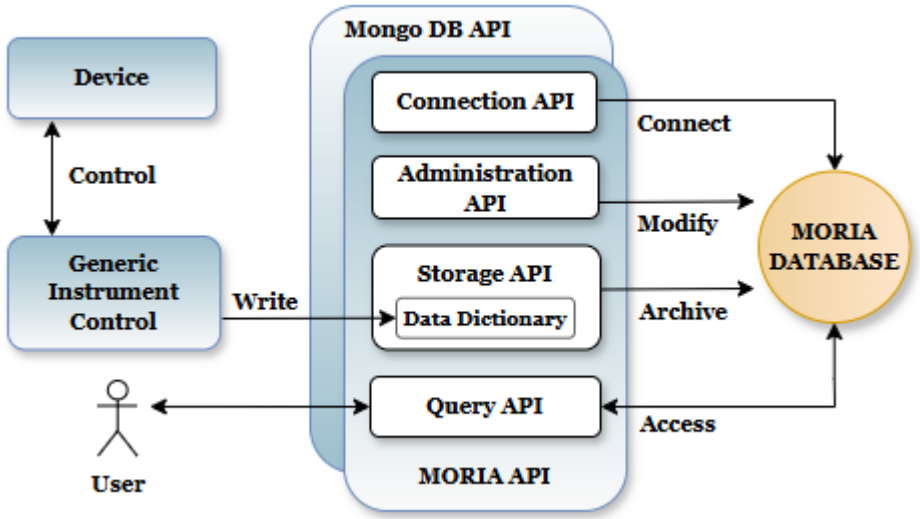
     Whenever possible, strive to define instruments and diagnostics in a generalizable manner, as each needs to be set only once. In our example, we follow a naming convention that uses uppercase letters with underscore delimiters for consistency; however, this is not strictly required.
   - **Example:**

     At GALADRIEL, a wide range of instruments and camera models operate concurrently during experiments. For instance, in one experiment, a gas jet was used to measure the electron spectrum emitted from its interaction at various positions. This setup required the use of multiple cameras—often of different models—as well as various motors to control the jet's position.

     In the metadata, the instrument field is consistently set to **CAMERA** for all cameras. Similarly, all motors responsible for controlling the jet's position are tagged as **STAGES**. The diagnostic field is assigned according to the device's function: cameras used to characterize the electron output are tagged as **ELECTRON_SPECTRUM**, while motors controlling the jet's position are labeled as **TARGET_POSITION**. Other devices may be assigned different tags depending on their specific roles.

     For more concrete examples, refer to **examples.py** in the moria repository.

4 MORIA API



The MORIA API serves as a comprehensive framework for interacting with the database. It abstracts much of the complexity of MongoDB syntax and aligns with the diagnostic-first data management approach. The specific API modules use can vary depending on the laboratory and the roles designated for internal and external users.

The API is organized into four core modules:

Administration API

- Enables users to modify database collections.
- Supports operations such as adding instruments and diagnostics, setting up experiments, and controlling the shot counter.
- Primarily intended for internal or administrative use.

Connection API

- Manages the core database connection.
- Automatically invoked by each of the other API modules.

Query API

- Provides an interface for querying the database.
- Allows users to retrieve data based on specified conditions.
- Typically used by both internal and external users.

Storage API

- Provides functionality for storing data in the database.
- Allows users to insert data into the acquisitions collection and GridFS.

This modular structure ensures flexibility, maintainability, and clarity for diverse use cases and user roles. Detailed examples using each module are available in **examples.py** within the moria repository.

4.1 Data Administration (Administration API)

4.1.1 Experimental Run Updates

Currently, only a limited set of global experimental parameters are stored in the `run_setup` collection. These parameters are accessible from any computer or program connected to the database server. Future parameters will likely be added.

The API provides several methods for updating these values as needed. To manage the shot number, use the `iterate_shot_number` function to increment the counter, or `reset_shot_count` to set it to zero. The `set_experiment` function is used to define the current experiment name. These values will be associated with all subsequently archived data.

4.1.2 Updating Data

The API includes functionality for both individual and bulk updates of documents within any collection in the database. This is handled by the `update_doc` function.

To perform an update, you must provide:

- A **filter criteria**: dictionary that specifies the key-value pairs used to identify the documents to update.
- An **update criteria**: dictionary that defines the changes to be applied.
- A **command**: this may include inserting new fields, replacing the values of existing fields, or appending to lists within existing fields.

Important: The update operation **can and will overwrite** existing field data and is **irreversible**. Use caution when specifying filter criteria. Filters should be as specific as possible—ideally targeting individual documents using identifiers such as **shot_number** or **device_name**—to avoid unintentionally modifying large portions of the database.

4.1.3 Deleting Data

Currently, there is no built-in feature for bulk data deletion from the database. This operation should be managed by your designated database administrator. Deleting data from a MongoDB database is a **non-recoverable operation**. Any potential damage caused by improper deletion can be mitigated with appropriate backup protocols—see the example in backup/restore procedures.

For now, data deletion is left to the discretion of users, as priorities may vary between laboratories. While this functionality could be implemented, no explicit function has been created for this purpose.

4.2 Archiving Data (Storage API)

Archiving data into the database is straightforward but requires minor data curation prior to archival. The `insert_data` function handles the archival of individual documents.

The data to be archived must follow a specific Python dictionary structure containing two sub-dictionaries: **data** and **metadata**.

4.2.1 Metadata Dictionary

The metadata dictionary contains indirect information related to the diagnostic measurement.

There are several **REQUIRED** fields that must be correctly populated for successful archival. These required fields vary depending on the target collection. For a detailed list of required fields by collection, see Tables and Fields.

4.2.2 Data Dictionary

The data dictionary holds the diagnostic data itself. This includes:

- Scalar or array measurements (`acquisitions`)
- Images or large output files (`fs.files/fs.chunks`)
- Processed images or derived data (`processed_data`)

The API automatically determines the appropriate storage location based on the contents of the dictionary. There is only one restriction on the contents of the data dictionary:

- If you are archiving direct image files (e.g., `.tiff`, `.pdf`, `.jpg`) or data exceeding **16 MB**, the image or data must be converted to a byte format and stored under the **buffer** field within the data dictionary. This convention signals the API to use GridFS for efficient storage.

4.3 Data Retrieval (Query API)

This section describes two methods for querying the database using the API, outlining the process and functionality of each approach.

4.3.1 Query Builder

The query builder is the primary method for querying data from the database. To execute a query, you must first construct it, a process that involves two main steps.

First, define the filter pipeline. This can be accomplished in two ways:

- By passing a dictionary containing one or more key-value pairs as an argument when calling `query_data_value`.
- By passing a dictionary containing one or more key-list pairs as an argument when calling `query_data_range`.

Notes:

- These functions can be called multiple times with different filter criteria. Each call is applied sequentially, allowing you to build up a more complex filter pipeline.
- Currently, users are required to know whether a field resides in the data or metadata sub-dictionaries to construct accurate queries, see examples in `query_data_range` and `query_data_value`.

  Since fields can be added dynamically, it is difficult to maintain a comprehensive list of all available queryable fields. The following methods can help identify them:

  - The fields stored under `data` are typically described in the `data_info` section of the metadata.
  - Metadata fields specific to each device can be retrieved using the `get_device_fields` function.
  - GUI tools such as MongoDB Compass can also be used to explore available fields and assist with query construction.
- To query fields located within subdictionaries of the `data` or `metadata` dictionaries, you must use the full dictionary path as the key.
  - For example, to perform a value-based query on the units field within the acquisitions collection, you would specify the key-value pair as: `'data_info.units' : ['fs ^2']`.
  - Note the use of a period (`.`) to separate the parent and subfield names in the key. This syntax follows MongoDB's dot notation for querying nested fields.

Once the filter pipeline is defined, execute the query by running `run_query`, which performs the aggregate MongoDB query. To clear the filter pipeline, use `clear_query`.

4.3.2 Direct GridFS Queries

The API also includes direct GridFS query functions, which are useful for retrieving quick, targeted information directly from GridFS without relying on the query builder. These queries are limited in scope and are executed independently. Available query methods include:

- `get_by_metadata`
- `get_by_filename`
- `get_in_time_range`

Additionally, the API provides several utility functions to assist with GridFS image handling and parsing of query results:

- `convert_gridfs_file_to_image`
- `get_metadata_from_cursor`

- image_to_nparray
- save_image_to_file
- view_image

These tools enable more flexible and efficient interaction with image data stored in the database.

5 Database Maintenance

5.1 Backup/Restore Procedures

The moria repository includes two Bash script utilities within **database_utils**, built on the MongoAPI, for database backup and restoration: `backup_database` and `restore_database`. Each script requires two arguments:

- **database_name**: The name of the database.
- **backup_directory**: The file path to the backup directory.

It is recommended to configure an automated backup script at a chosen interval to ensure database recoverability in the event of unintended deletions, database failures, or corruption.

6 Appendices

6.1 Tables and Fields

The following provides detailed information on each collection in the database. Only the required fields for each collection are outlined below; additional fields and data can be included as needed without issue.

run_setup

Contains global information about the database and current experimental setup.

Required fields:

last_shot : *int*
Global database shot counter denoting the last shot iterated.

experiment : *str*
Name of the current experiment.

Example:

```
{ _id: ObjectId("65568cef2bd6dcb2cbe45b10"),
  last_shot: 99999,
  experiment: 'TEST_EXPERIMENT'
}
```

diagnostics

This collection contains a document for each diagnostic in your system.

Required fields:

_id : *Mongo - ObjectId*, (automatic)
Auto-generated unique document ID.

diagnostic : *str*
The diagnostic name. Should correspond to a general name for a commonly used diagnostic.

Examples:

```
{ _id: ObjectId("6780557d72404c3088a95ca6"),
  diagnostic: 'LASER_ENERGY'
},
{ _id: ObjectId("6786ae8a72404c3088a95ca7"),
  diagnostic: 'ELECTRON_SPECTRUM'
}
```

instruments

This collection contains a document for each instrument in your system.

Required fields:

_id : *Mongo - ObjectId*, (automatic)
Auto-generated unique document ID.

instrument : *str*, (required)
The instrument name. Should correspond to a general purpose classification of instrument type.

gridfs : *bool*, (required)
If true, any data with tag with this instrument name will be archived in GridFS.

Examples:

```
{ _id: ObjectId("6772d794895cfbfc85bfd71f"),
  instrument: 'PHASE_CONTROLLER',
  gridfs: false
},
{ _id: ObjectId("677c5df472404c3088a95ca3"),
  instrument: 'CAMERA',
  gridfs: true
}
```

fs.files

This collection contains the metadata documents corresponding to the data stored in fs.chunks.

Required fields:

_id : *Mongo - ObjectId*, (automatic)
Auto-generated unique document ID, this is required to reassemble the sharded data in **fs.chunks**.

filename : *str*, (automatic)
File name.

metadata : *dict*
Contains the metadata relating to a single laser pulse.

shot_number : *int*
Shot number corresponding to a single laser pulse. **IMPORTANT**: must be unique.

experiment : *str*
Experiment name.

trigger_timestamp : *Python - ISODate*
Time when the acquisition is initiated.

archive_timestamp : *Python - ISODate*
Time when the data is archived.

instrument : *str*
The instrument name. Should correspond to a general purpose classification of instrument type.
IMPORTANT: must exist in instruments.

diagnostic : *str*
The diagnostic name. Should correspond to a general name for a commonly used diagnostic.
IMPORTANT: must exist in diagnostics.

device_name : *str*
Individual device identifier. **IMPORTANT**: must be unique.

data_info : *dict*
This is empty in GridFS.

notes : *list*
List containing individual document comments, can be added to using `add_note_to_doc`.

chunksize : *int*, (automatic)
The size of each chunk in bytes.

length : *long*, (automatic)
The size of the document in bytes.

uploadDate : *Python — ISODate*, (automatic)
Date time the file was uploaded to GridFS.

Example:

```
{
    _id: ObjectId("67ec2115c80e92e14e4c4d07"),
    filename: 'basler_d20250401_t102332807490.tif',
    metadata: {
      shot_number: 99999,
      experiment: 'TEST_EXPERIMENT',
      trigger_timestamp: ISODate("2025-04-01T10:23:32.779Z"),
      archive_timestamp: ISODate("2025-04-01T10:23:33.984Z"),
      instrument: 'CAMERA',
      diagnostic: 'LASER_ENERGY',
      device_name: 'basler_0',
      data_info: {
        image: {
          data_type: 'GridOut',
          units: '',
          description: 'recorded image sharded by GridFS'
        }
      },
      notes: [],
      exposure_time: 90000,
      pixel_depth: 'Mono10',
      model: 'acA1300-200um',
      manufacturer: 'Basler',
      file_name: 'basler_d20250401_t102332807490.tif'
    },
    chunkSize: 261120,
    length: Long("2621562"),
    uploadDate: ISODate("2025-04-01T17:23:34.002Z")
}
```

acquisitions

This collection contains the data and metadata pertaning to raw data acquisiton.

Required fields:

_id : *Mongo - ObjectId*, (automatic)
Auto-generated unique document ID.

data : *dict*
Contains individual device data for a single laser pulse.

metadata : *dict*
Contains the metadata relating to a single laser pulse.

shot_number : *int*
Shot number corresponding to a single laser pulse. **IMPORTANT**: must be unique.

experiment : *str*
Experiment name.

trigger_timestamp : *Python - ISODate*
Time when the acquisition is initiated.

archive_timestamp : *Python - ISODate*
Time when the data is archived.

instrument : *str*
The instrument name. Should correspond to a general purpose classification of instrument type.
IMPORTANT: must exist in instruments.

diagnostic : *str*
The diagnostic name. Should correspond to a general name for a commonly used diagnostic.
IMPORTANT: must exist in diagnostics.

device_name : *str*
Individual device identifier. **IMPORTANT**: must be unique.

data_info : *dict*
Contains one dictionary that describes each field within the data dictionary.
field1_name :
data_type : *str* (int, float, string, array[data type], GridOut)
units : *str*
description : *str*
etc...

notes : *list*
List containing individual document comments, can be added to using `add_note_to_doc`.

Example:

```
{
    _id: '65a36c2b-d295-4d11-9993-67b7fb77ec0c',
    data: {
      delay: 4400,
      order2: 35324.962946695436,
      order3: 19862.21681901822,
      order4: -3969393.253744395
    },
    metadata: {
      shot_number: 99999,
      experiment: 'TEST_EXPERIMENT',
      trigger_timestamp: ISODate("2025-04-01T10:23:14.842Z"),
      archive_timestamp: ISODate("2025-04-01T10:23:33.938Z"),
      instrument: 'PHASE_CONTROLLER',
      diagnostic: 'SYSTEM',
      device_name: 'dazzler',
      data_info: {
        delay: {
          data_type: 'float',
          units: 'fs',
          description: 'first order of the spectral phase expansion'
        },
        order2: {
          data_type: 'float',
          units: 'fs^2',
          description: 'second order of the spectral phase expansion'
        },
        order3: {
          data_type: 'float',
          units: 'fs^3',
          description: 'third order of the spectral phase expansion'
        },
        order4: {
          data_type: 'float',
          units: 'fs^4',
          description: 'forth order of the spectral phase expansion'
        }
      },
      notes: [],
      manufacturer: 'fastlite'
    }
}
```

processed_data

Contains information not pertaining to a single device acquisition.

Required fields:

_id : *Mongo - ObjectId*, (automatic)
Auto-generated unique document ID, this is required to reassemble the sharded data in **fs.chunks**.

filename : *str*, (automatic)
File name.

metadata : *dict*
Contains the metadata relating to a single laser pulse.

shot_number : *int*
Shot number corresponding to a single laser pulse. **IMPORTANT**: must be unique.

experiment : *str*
Experiment name.

archive_timestamp : *Python - ISODate*
Time when the data is archived.

diagnostic : *str*
The diagnostic name. Should correspond to a general name for a commonly used diagnostic.
IMPORTANT: must exist in diagnostics.

OR

process : *str*
The process or analysis name.

data_info : *dict*
Contains one dictionary that describes each field within the data dictionary.
field1_name :
data_type : *str* (int, float, string, array[data type], GridOut)
units : *str*
description : *str*
etc...

notes : *list*
List containing individual document comments, can be added to using add_note_to_doc.

chunksize : *int*, (automatic)
The size of each chunk in bytes.

length : *long*, (automatic)
The size of the document in bytes.

uploadDate : *Python — ISODate*, (automatic)
Date time the file was uploaded to GridFS.

Example:

```
{
    _id: '15a48810-3726-4d4c-b602-f67a300d36fe',
    data: { raw_pixel_sum: 11475639, laser_energy: 1353202.7528855524 },
    metadata: {
      shot_number: 99999,
      experiment: 'TEST_EXPERIMENT',
      archive_timestamp: ISODate("2025-04-01T10:23:34.065Z"),
      diagnostic: 'LASER_ENERGY',
      data_info: {
        raw_pixel_sum: {
          data_type: 'int',
          units: '',
          description: 'Sum of pixel values without subtraction'
        },
        laser_energy: {
          data_type: 'float',
          units: 'Joules',
          description: 'Laser energy calculated from camera'
        }
      },
      notes: [],
      acq_device: [ 'basler_0' ]
    }
}
```

6.2 MORIA API Detail

This section contains a detailed overview of each function in the MORIA API library.

6.2.1 Administration API

add_diagnostic(*diagnostic*)

Add a new diagnostic to the diagnostics collection.

Parameters:

diagnostic : *str*
The diagnostic name. Should correspond to a general name for a commonly used diagnostic.

add_instrument(*instrument*, *gridfs_bool*)

Add a new instrument to the instruments collection.

Parameters:

instrument : *str*
The instrument name. Should correspond to a general purpose classification of instrument type (i.e. CAMERA).

gridfs_bool : *bool, default = None*
If True, the documents the are archived will be stored using GridFS. The data will be sharded and stored in the fs.chunks collection and the metadata will be archived in the fs.files collection and will contain information to reassemble the sharded chunks.

add_note_to_doc(*shot_number*, *device_name*, *msg*)

Add a note to a single documents notes list. It also stores the time this note was created and who it was created by automatically.

Parameters:

shot_number : *int*
The unique shot number.

device_name : *str*
The unique device name.

msg : *str*
The note content.

iterate_shot_number()

Adds one to the *last_shot* field in the run_setup collection.

remove_diagnostic(*diagnostic*)

Remove an diagnostic from the diagnostics collection.

Parameters:

instrument : *str*
The name of a valid diagnostic stored in the diagnostics collection.

remove_instrument(*instrument*)

Remove an instrument from the instruments collection.

Parameters:

instrument : *str*
The name of a valid instrument stored in the instruments collection (i.e. CAMERA).

reset_shot_count()

Sets the *last_shot* field in the run_setup collection to zero.

set_experiment(*name*)

Sets the name of the experiment in the run_setup collection. Once set this field can be queried from anywhere and archived within your metadata dictionary.

Parameters:

name : *str*
Experiment name.

update_doc(*update_dict*, *value_filter_dict*, *range_filter_dict*, *command*)

This function modifies documents that match the criteria defined by the value and filter dictionaries, updating the specified fields according to the key-value pairs provided in the update dictionary.

IMPORTANT: this function can overwrite data.

Parameters:

update_dict : *dict*
A dictionary containing the updated values for the specified fields.

value_filter_dict : *dict*, *default* = {}
A dictionary containing key-value pairs used to isolate the documents targeted for modification. See query_data_value.

range_filter_dict : *dict*, *default* = {}
A dictionary containing key-range pairs used to isolate the documents targeted for modification. See query_data_range.

command : *str*, *default* = *'replace'*
replace
This operation replaces the value in the document with the corresponding value from the update dictionary, based on matching field names.

insert
This operation adds a new field-value pair to the specified document(s). If the field already exists, its current value will be overwritten.

append
This operation will append the value(s) from the update dictionary to the corresponding dictionary or list field in the target document(s), based on matching field names. If the specified field does not exist, it will be created.

6.2.2 Connection API

Database(*db_name*, *server_ip)*

Initializes a connection to a live MongoDB service, allowing for querying and archiving.

Parameters:

db_name : *str*
The name of the database on the corresponding server.

server_ip : *str*, *optional*, *default* = *None*
The IP address of the server that the data is stored on.

Returns:

Object containing the database connection via MongoClient.

Example:

```
from database import Database, StorageAPI, QueryAPI, AdminAPI

db = Database("test_db", None)
storage = StorageAPI(db)
query   = QueryAPI(db)
admin   = AdminAPI(db)
```

6.2.3 Query API

clear_query()

Removes the current query filters from the pipeline.

convert_gridfs_file_to_image(*gridfs_file*)

Retrieves the image from GridFS file.

Parameters:

gridfs_file : *GridOut file object*

Returns:

image : *Image object*
The image in **TIFF** format.

diagnostic_exists(*diagnostic*)

Checks if a diagnostic exists in the *diagnostics* collection.

Parameters:

diagnostic : *str*

Returns:

num_docs : *int, [0 or 1]*
0 if it is not found, **1** if it is found.

get_by_filename(*filename*)

Retrieves the GridFS files matching the filename field in the metadata.

Parameters:

filename : *str*

Returns:

gridfs_file : *GridOut file object*

get_by_metadata(*key*, *value*)

Retrieves the GridFS files matching the metadata key-value pair.

Parameters:

key : *str*
Metadata tag (field).

value : *any*
Value requested.

Returns:

gridfs_files : *list*
Contains the GridFS files that match the query.

get_device_fields(*device_name*)

Retrieves a single devices' metadata fields corresponding to the last stored document.

Parameters:

device_name : *str*
The name of the specific device. Corresponds to the field 'device_name' in the metadata part of the document.

Returns:

field_list : *list*
Complete list of the metadata tags.

get_experiment()

Retrieves the current experiment name from the run_setup collection.

get_in_time_range(*datetime_a*, *datetime_b*)

Retrieves the GridFS files that were acquired within a certain time frame.

Parameters:

datetime_a : *str*
Beginning date/time in string format.

datetime_b : *str*
Ending date/time in string format.

Returns:

gridfs_files : *list*
Contains the GridFS files that match the query.

get_last_shot_number()

Retrieves the last shot number from the *run_setup* collection.

get_metadata_from_cursor(*gridfs_file*)

Retrieves the metadata associated with the GridFS file.

Parameters:

gridfs_file : *GridOut file object*

Returns:

metadata : *dict*
The metadata dictionary from fs.files associated with the GridFS file.

hardware_info(*dev_type*, *print_list*)

Retrieves the list of instruments **OR** diagnostics currently set for data storage from their respective collections.

Parameters:

dev_type : *str, "instruments" or "diagnostics"*
The collection you want to search.

print_list : *bool, optional*
If you want to print the output in the terminal.

Returns:

instrument/diagnostic : *list*
List containing the requested devices.

image_to_nparray(*gridfs_file*)

Converts the image stored in the GridFS file into a numpy array.

Parameters:

gridfs_file : *GridOut file object*

Returns:

array : *ndarray*
Contains the image pixel values.

instrument_exists(*instrument*)

Checks if the instrument exists in the *instruments* collection.

Parameters:

instrument : *str*

Returns:

num_docs : *int, [0 or 1]*
0 if it is not found, **1** if it is found.

instrument_in_gridfs(*instrument*)

Checks if an instrument is set to archive using GridFS.

Parameters:

instrument : *str*

Returns:

grid_fs : *bool*
The archive status that was set when the instrument was added.

query_data_range(*query_dict*)

This function parses the input dictionary and adds each key-range pair to the filter pipeline. It is capable of handling both numerical and time ranges. This function is used in conjunction with `run_query`.

Note: Range queries must be numeric list with a start and end.

Parameters:

query_dict : *dict*

Example input dictionary:

```
range_query_dict = { 'metadata' : {'shot_number' : [<lower bound>, <upper bound>]},
                     'data' : {'data_field' : [<lower_bound>, <upper_bound>}
}
```

query_data_value(*query_dict*)

This function parses the input dictionary and adds each key-value pair to the filter pipeline. This function is used in conjunction with `run_query`.

Note: Value queries must be a list even if only one value is requested.

Parameters:

grid_fs : *bool*
The archive status that was set when the instrument was added.

Example input dictionary:

```
value_query_dict = { 'metadata' : { 'device_name' : [device_name],
                                    'shot_number' : [1]
                                  }
                   }
```

run_query(*fetch_related_data*, *make_dict*)

Executes a MongoDB aggregate query utilizing the current requests in the filter pipeline.

Parameters:

fetch_related_data : *bool, default = False*
If enabled, returns all data associated with the shots that match the filters applied to each collection individually.
If this argument is not enabled, the function returns only the data from documents with the requested filters.

make_dict : *bool, default = True*
If enabled, returns a user-friendly dictionary, organizing the matching documents by device_name.
If this argument is not enabled, the function returns a MongoDB cursor object, which must be parsed manually.

NOTE: if you need faster querying set this argument to False.

save_image_to_file(*gridfs_file*)

Saves an image and its associated metadata to the current directory.

Parameters:

gridfs_file : *GridOut file object*

view_image(*gridfs_file*)

Coverts the gridfs file to an image and displays it.

Parameters:

gridfs_file : *GridOut file object*

6.2.4 Storage API

insert_data(*data_struct, processed_dict)*

Archive a single document into the database. This function will dynamically determine where the data needs to be archived in based on the preference set when the instrument was added to the instruments collection.

Parameters:

data_struct : *dict*
This structure consists of layered dictionaries that contain both the measurement data and associated metadata for a single device.
A document will only be archived if all required fields are present in this structure. See Tables and Fields for specific collection requirements.

processed_dict : *bool, default = False*
If this flag is enabled, the data will be stored in the processed_data collection instead of the raw data acquisitions collection or GridFS.