

# Documentation

# Table of contents

[Home](#)

[About](#)

[Architecture](#)

[Documentation](#)

[TreeWalk](#)

[Database](#)

[MdH-WebUI/Query-Server](#)

[WebUI](#)

[Installation](#)

[Usage](#)

[FAQ](#)

# About



## Metadata-Hub

This application is developed in the course of the [AMOS](#) project.

## Motivation

---

Big-Data environments often comprise large amounts of data that have been integrated from various sources. These can only be turned into valuable information through the use of metadata, which is significantly more lightweight, allowing for faster accessing and handling when compared to the actual data. Therefore, the aim of the Metadata-Hub is to provide a platform-independent retrieval, storage, and query mechanism for metadata on large file systems. This will allow end-user applications to obtain resilient and meaningful data as basis for complex data analyses in order to mine valuable information for the application-specific context.

## Goals

---

Our mission is to create a first **prototype** of the Metadata-Hub, an independent piece of software that intelligently crawls large file systems in order to gain and store metadata about the files it finds. An intelligent algorithm continuously traverses the file system and collects interesting metadata. This metadata is stored in a designated metadata store, thereby generating an easy-to-access and persistent index of the whole file system. Finally, existing end-user applications will be able to query and consume the collected metadata.

## Wiki

---

This wiki is the official documentation of the Metadata-Hub project. All included images are preview images that are linked to the larger original image. Simply click on the image to view it in higher resolution.

# Architecture

This chapter provides a short overview about the conceptual architecture of the Metadata-Hub application. It aims to give a high-level understanding of the structure of the application rather than a detailed description of its components. For a more detailed documentation, please refer to the [Documentation](#) chapter.

The Metadata-Hub application consists of three major components that are listed below.

- **TreeWalk**

This component implements the TreeWalk algorithm which crawls the target directories/filesystems and stores the by the [ExifTool](#) extracted metadata in the database. The TreeWalk supports scheduled and periodic executions and controlling mechanism such as pausing, continuing and stopping running executions.

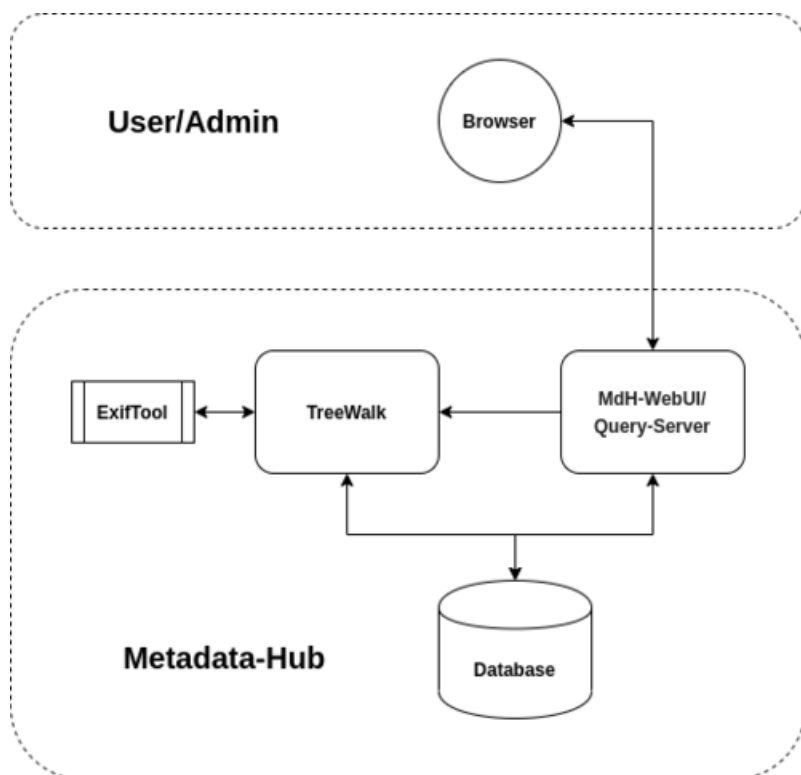
- **MdH-WebUI/Query-Server**

The MdH-WebUI/Query-Server provides the interface for metadata queries and the controlling of the TreeWalk. It uses [GraphQL](#) as the query language. Furthermore, it supports queries to be stored and executed again as well as exporting the retrieved results as JSON files.

- **Database**

The metadata and additional stateful information for the TreeWalk is stored in a [PostgreSQL](#) database.

The interaction of these components is depicted in the following graphic. The user/admin use the web interface provided by the MdH-WebUI/Query-Server to query metadata and configure the TreeWalk.



# Documentation

This chapter provides a more detailed overview about the single components. It aims to give a deeper understanding of how they work and what functionality they provide. Though, this is no source code documentation. Therefore, please refer to the [GitHub repository](#).

# TreeWalk

## WebUI

---

This section will shortly explain the interface of the TreeWalk component. Metadata-Hub provides a web interface for using the TreeWalk component:

- **Controller**  
Interface for controlling (starting/stopping/etc.) the TreeWalk.
- **Schedule**  
Interface for viewing/modifying the schedule of the TreeWalk.
- **Intervals**  
Interface for viewing/modifying the intervals of the TreeWalk.

## Configuration

The best way to configure a TreeWalk execution is to use the web interface. Here is a short summary about how to configure an execution. These help messages are also displayed on the web interface.

- **Name**  
The name of the configuration. It should be a short name that describes the purpose of the configuration.
- **Author**  
The author of the configuration.
- **Description**  
A more detailed description about the configuration. This field is optional.
- **Start**  
Start timestamp of the configuration. It must be according to the format `YY-MM-DD hh:mm:ss` e.g. `2020-07-22 10:15:00`.
- **Interval**  
This defines the interval in which the configuration is executed periodically. Input the number of hours and days the in which the execution should repeated. If the configuration should only be executed once, leave both values as `0`.
- **Directories**  
This field expects a path to a directory and a corresponding boolean separated by a comma. This value determines, if the tree walk is supposed to be executed recursively (If the value is set to true, all subdirectories will be scanned). You can also enter multiple pairs, by separating them with a semicolon.

An example input could look like this:

```
/home/metadata-hub, True; /home/user, false
```

- **ExifTool**  
This field is used to choose between the linux and windows executables of the ExifTool. Pick the operating system you are currently working on.
- **CPU-Level**  
This field is used to determine how many system resources should be reserved for the tree walk execution. `1` provides the system with the minimum amount, `4` with the maximum.
- **Package-Size**  
This setting defines how many files are combined in one work package for processing. Directories with a large amount of files are split up and directories with a small amount of files are combined according to this value in order to use the resources more efficiently. A very small value for this setting results in TreeWalk being more responsive, e.g. stopping the TreeWalk during execution will be faster. On the other hand, performance will be slowed down due to additional overhead. A large value will result in an opposite behaviour. Please provide a number between `10` and `1000` here. A reliable default value is `100`.
- **Force-Update**  
This field determines, if the submission that is about to be sent to the TreeWalk will replace the old one. If `no` is selected the TreeWalk will append this execution to the schedule.

If you plan to configure the TreeWalk without the web interface, please have a look at the schema defining the JSON configuration. This is required for accessing the API of the TreeWalk directly.

## Examples

This section will show example pictures of the web interface. It will be updated until 22nd July.

## Architecture

---

The TreeWalk is split into four major components. The abbreviaton *TW* stands for *TreeWalk*.

- **TWManager**

The manager controls the state of the TreeWalk. In detail, it controls the worker processes and executes control mechanisms such as pausing or stopping the TreeWalk.

- **TWScheduler**

The scheduler keeps track of scheduled executions and time interval restrictions. It dispatches executions to the TWManager when they are ready to run and updates their status in the database.

- **TWApi**

The API provides a simple REST interface for accessing the functionality of the TreeWalk. It's implemented as a simple Flask application.

- **TWDatabaseUpdater**

The database updater periodically removes files in the database that were marked as deleted but still have a certain timeout before they are totally removed from the storage.

- **TWWorker**

The worker runs the ExifTool, creates hashes of the single files and inserts the results into the database.

All these components are implemented as separate threads except the workers that run as standalone processes. They all run in `while True` loops but block on certain events to reduce CPU usage. For example, the manager is simply blocking execution until a new command is retrieved when the TreeWalk is not running.

## Algorithm

---

This is just a small sketch about the steps that are executed when the TreeWalk is running.

1. The input directories are traversed in order to create *evenly* distributed work load for the workers. Packages are created in a way that directories containing few files are combined and directories that contain many files are split up at a later point.
2. While running, the manager dispatches each worker a new package. The workers then run the ExifTool and calculate the file hashes. Afterwards, they insert the data into the database. The worker that finished last signals the manager that the iteration has finished.
3. The manager then updates the progress that have been made and checks if a new command has been send in the meantime. If this is the case, the manager exeuctes the retrieved command, e.g pausing the TreeWalk. When no command was sent, it simply dispatches each worker a new work package.
4. When no work packages are present anymore, all data strucuteres are cleared and the worker processes terminate. Finally, the state changes to ready again.

## API

---

This section will shortly explain the API of the TreeWalk. This is helpful when you plan to use the TreeWalk as a standalone component. For the sake of simplicity, all these endpoints support `GET` and `POST` requests except `/info` that only supports `GET` requests.

- **/info**

- Description:

- Retrieve information about the current status of the TreeWalk.

- Parameters:

- None

- **/start**

- Description:  
Start the TreeWalk with a certain configuration. If the execution cannot be started immediately, it is appended to the schedule.
- Parameters:
  - `config` (*required*)  
This is the configuration of the execution. It can either be a filepath pointing to a valid configuration file or the string representation of a valid JSON configuration.
  - `update` (*optional*)  
By providing `update=True`, a possible running execution will be stopped and the new one will be started. Without providing `update` or with `update=False`, the request will be appended to the schedule if an execution is currently running/paused.

- **/pause**

- Description:  
Pause a currently running execution of the TreeWalk. The request will be ignored when the tree walk is currently not running. The execution can be continued or stopped later on.
- Parameters:  
None

- **/continue**

- Description:  
Continue a paused execution of the TreeWalk. The request will be ignored if the TreeWalk is not paused.
- Parameters:  
None

- **/stop**

- Description:  
Stop a running or paused execution of the TreeWalk. The request will be ignored if the TreeWalk is ready. The execution will be marked as aborted in the database.
- Parameters:  
None

- **/schedule/list**

- Description:  
List the current schedule formatted as JSON.
- Parameters:  
None

- **/schedule/remove**

- Description:  
Remove an entry of the schedule. On success, the corresponding (periodical) execution is removed from the schedule. Otherwise, an error message is returned.
- Parameters:
  - `id` (*required*)  
The unique identifier of the entry. The identifier can be retrieved from `schedule/list` for example.

- **/intervals/list**

- Description:  
List all existing intervals for restricting maximum resource consumption.
- Parameters:  
None

- **/intervals/add**



- Description:  
Add an interval for restricting maximum resource consumption. If an interval is invalid or conflicts with an already existing one, it is not added. The timestamps described in the parameter section must be of the form `dd:hh:mm` with `dd` is between `00` and `06` (specifying the weekday), `hh` between `00` and `23` (specifying the hours) and `mm` between `00` and `59` (specifying the minutes).

- Parameters:

- `start` (*required*)  
Start timestamp of the interval.
- `end` (*required*)  
End timestamp of the interval.
- `cpu` (*required*)  
Maximum CPU level during the interval. Must be `1`, `2`, `3` or `4`.

- **/schedule/remove**

- Description:  
Remove an interval. On success, the corresponding interval is removed. Otherwise, an error message is returned.

- Parameters:

- `id` (*required*)  
The unique identifier of the interval. The identifier can be retrieved from `intervals/list` for example.

- **/shutdown**

- Description:  
Shutdown the crawler completely. This will force a possible running execution of the TreeWalk to end and exit the TreeWalk process.

## CLI

The TreeWalk can be also controlled via the command line. The module `cli.py` forwards the commands to the REST API of the TreeWalk and prints the response to the console. It can be directly executed from the repository. In this case, the default environment settings `configs/environment.default.json` will be used. For using another setting, either change the values in this file or let the environment variable `METADAHUB_ENV` point to the file with the correct settings. Please have a look at the [schema](#) of a TreeWalk configuration and the [environment settings](#) before. When everything was set up properly and the TreeWalk is running, an exemplary usage would look like this:

```
$ python3 cli.py info
{
  "status": "ready",
  "config": null,
  "processes": 0
}
$ python3 cli.py stop
Attempted to stop when TreeWalk was ready.
```

## Evaluations

In this section, time measurements / benchmarkings of the TreeWalk are discussed. *TW* is a abbreviation for *TreeWalk*.

### sprint-10-release vs. separate-database-threads

We shortly evaluated two versions of the implementation of the TreeWalk in order to decide which one to include in the final release of the product.

- **sprint-10-release**

In this version, TWWorkers (separate processes) run the ExifTool, file hashing and database operations. When running, the TWManager dispatches a work package to each TWWorker and waits for all TWWorkers to finish before continuing with the next one.

- **seperate-database-threads**

In this version, TWWorker (separate processes) run only the ExifTool and file hashing. The database operations are dispatched to dedicated database threads. Before running, the TWManager dispatches all work packages to the TWWorker.

Based on this behaviour, we thought the version *seperate-database-threads* would lead to an performance improvement because the database operations don't block the TWWorker anymore, reducing waiting time due to the database table locking. Unfortunately, the opposite happened. We ran some test scenarios and already noticed a huge difference on small datasets.

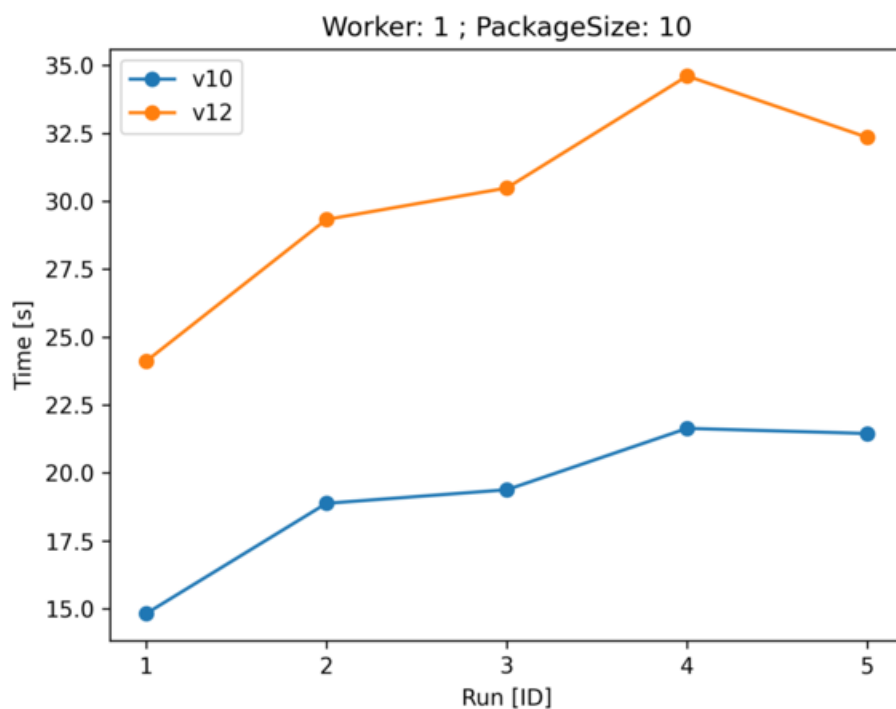
Technical setup:

- OS: Ubuntu 20.04
- Python: 3.8
- The PostgreSQL instance was running in a Docker container.
- Both the database storage and test directory were located at an external USB drive with sequential read ~150 MB/s and write ~40 MB/s measured by the Gnome Disk benchmarking utility in idle state
- CPU: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
- RAM: 2x 8GB DDR4 2400 MT/s
- General system load (background applications, etc.) were set to a minimum

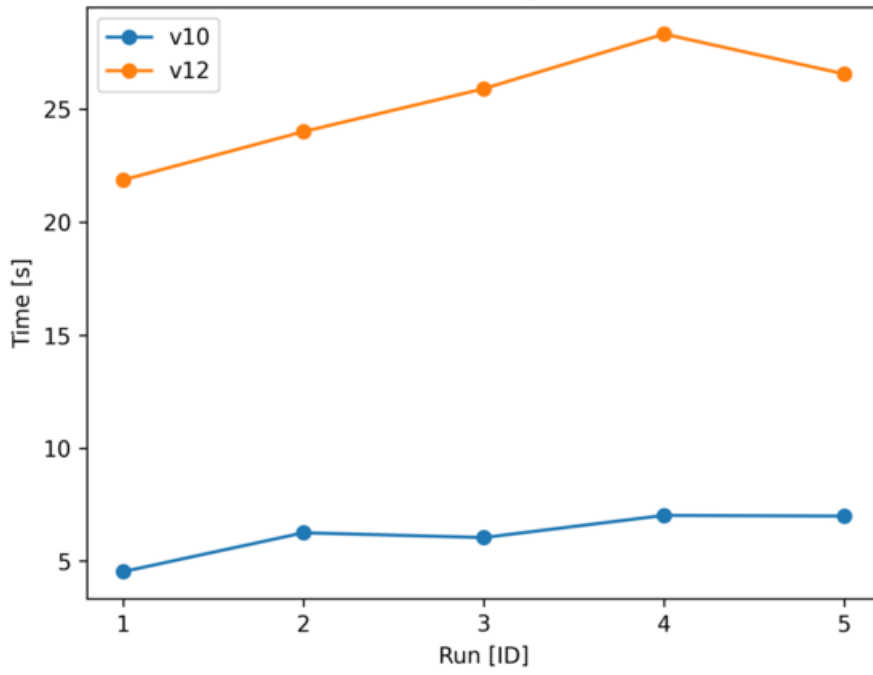
The first measurement crawled the directory `reference_tree` (1441 files, ~601 MB).

- Worker: number of TWWorker
- Package-Size: max. number of files in one work package
- Run: for each worker/package-size combination, 5 executions were ran

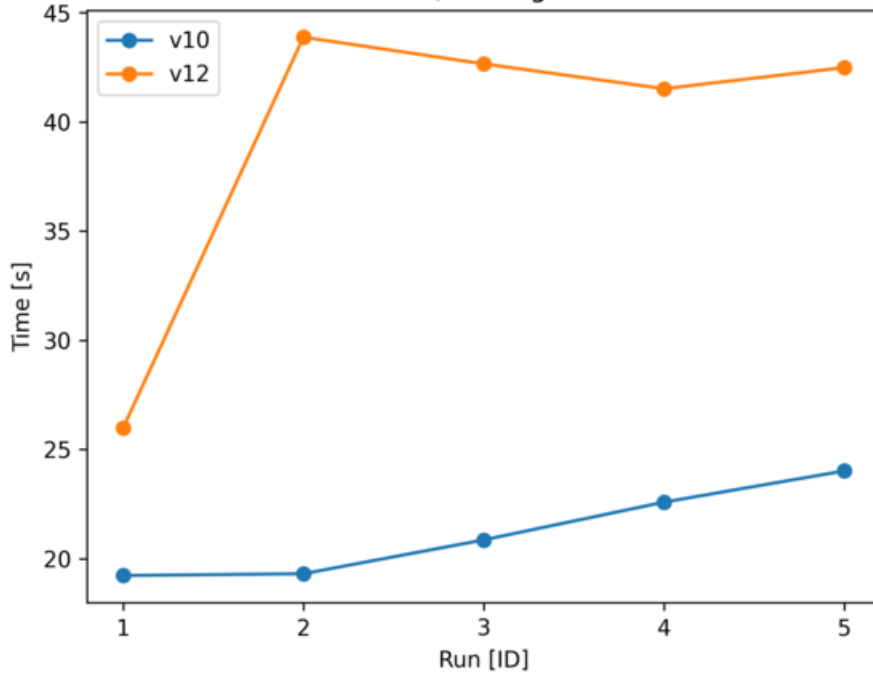
After the 5 subsequent runs of one configuration & version, the database was deleted and restarted. Here are the results with *v12* being the *seperate-database-threads* version and *v10* the *sprint-10-release*.

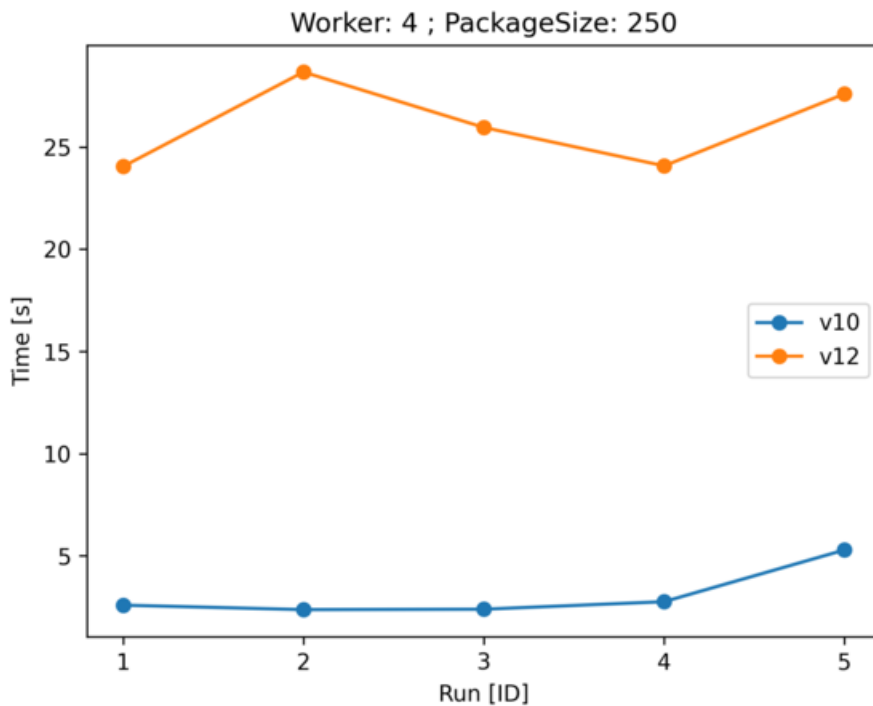


Worker: 1 ; PackageSize: 250



Worker: 4 ; PackageSize: 10





It shows that the *sprint-10-release* version outperforms the *separate-database-threads* version. There could be several explanations for this:

- The database operations aren't really bottlenecking the executions of the TWWorker. Of course, this heavily depends on where the database is located and *how fast* it operates.
- Additional IPC overhead in the *separate-database-threads* is higher than expected. In this version, multiple queues were added to enable the TWWorker to exchange data with the database threads.
- Synchronization overhead. This was actually the reason to improve the *sprint-10-release* version, but the more complex design of *separate-database-threads* also requires some additional more syncing between the different processes, threads, etc. It mainly should be relevant for actions controlling the TreeWalk, such as pausing, stopping, etc. but possible bugs and a poor implementation could interfere here.
- In the *separate-database-threads* version, it could be possible that the TWWorker produce too much work for the database threads to handle. This would lead to the situation in which database operations run *sequentially* again, but with the additional IPC & syncing overhead.

Another one-run example also shows the difference between these versions. It was run on a dataset with .mp4 files, 4687 in total with a size of ~9GB. The technical setup was the same.



In this example, the version *sprint-10-release* is also faster in execution. Based on this observation, we just ran this version on a larger dataset. The dataset **Dataset** consists of 1.177.441 files with a total size of ~35GB. It was located at an other external drive.

The execution took about 2 hours and 32 minutes, where 1 hour and 21 minutes were spent in running the ExifTool (roughly). The problem that occurred was investigated with **iostat** were read access decreased to ~2-5 MB/s during the measurement for most of the time. This would also explain the huge difference in the execution time of ~3-4 minutes for a 9GB dataset.

# Database

## Database

The database Metadata-Hub is using is a [PostgreSQL](#) database. It stores the extracted metadata of the crawled files as well as auxiliary data for the TreeWalk, Server and WebUI.

### Database Setup

1. Create the database role - [metadatabase-role.sql](#)
2. Create the database - [metadatabase-database.sql](#)
3. Import the schema - [metadatabase-schemata.sql](#)

Default setup:

- "database-name": "metadatabase",
- "database-host": "localhost",
- "database-port": 5432,
- "database-user": "metadatabase",
- "database-password": "metadatabase",

If another setup wants to be used, the `.sql` files in `metadatabase-hub/database` needs to be adapted.

### Loading the Schema:

psql command for Linux

```
psql metadatabase -U metadatabase -W -f metadatabase-schemata.sql
```

psql command for Windows

```
metadatabase> \i c:/dir/dir/metadatabase-schemata.sql
```

### Database-Schema

The database uses the following tables:

- crawls
- files
- metadata
- intervals
- stored\_editor\_queries
- schedule
- file\_categories

There's a short description about each of these tables in the upcoming sections.

### crawls

This table stores information about executions of the TreeWalk. It updates the state, e.g *paused* or *aborted*, and finishing times upon completion of the execution.

column	type	description
--------	------	-------------

column	type	description
id	bigint	unique identifier of the execution
dir_path	text	list of all input directories separated by ','
author	text	author of the configuration
name	text	name of the configuration
status	text	status of the execution
crawl_config	text	configuration of the TreeWalk execution
starting_time	timestamp with time zone	start of the execution
finished_time	timestamp with time zone	end of the execution
update_time	timestamp with time zone	timestamp when the corresponding was updated the last time

## files

This table stores information about files and their metadata. It is filled and updated by the crawler and is queried by the server.

column	type	description
id	bigint	unique identifier of the file
crawl_id	bigint	unique identifier of the execution
dir_path	text	the directory path to the file without it's name
name	text	file name
type	text	file type
size	bigint	file size in bytes
metadata	jsonb	this file's metadata attributes and their values
creation_time	timestamp with time zone	creation time of the file
access_time	timestamp with time zone	last time the file was accessed
modification_time	timestamp with time zone	last time the file was modified
file_hash	text	sha256-hash of the file
deleted	boolean	flag signaling, if the file was deleted since the last crawl
deleted_time	timestamp with time zone	time when the file was found to be deleted
in_metadata	boolean	flag signaling, if the file's metadata information was already added to the metadata table

## metadata

This table stores all the file types present in the files table, and their associated tags. It is used in the UI for file type and metadata attribute autocomplete and setting the datatype of a metadata attribute.

column	type	description
file_type	text	file type present in the files table
tag	json	the file type's associated tags, their number of occurrence and the datatype of the tag (String or Digit)

## intervals

This tables stores information about the time intervals of the TreeWalk that restrict maximum CPU consumption.

column	type	description
id	text	unique identifier of an interval
start_time	text	start time of the interval
end_time	text	end time of the interval
cpu_level	bigint	maximum CPU level during the interval

**stored\_editor\_queries**

This table stores information about queries created by the Query Editor in the WebUI. In the WebUI they can be saved and restored again.

column	type	description
id	bigint	unique identifier of a query
author	text	author of the query
title	text	name of the query
create_time	timestamp with time zone	creation time of the query
data	jsonb	data which is used to restore the query in the Query Editor

**schedule**

This tables stores information about scheduled (periodically) TreeWalk executions.

column	type	description
id	text	unique identifier of an entry
config	json	configuration of the TreeWalk execution
timestamp	timestamp without time zone	timestamp of next execution
force	boolean	true if the execution should force an already running one to stop
pending	boolean	true if the execution is already pending
interval	bigint	interval in seconds in which the execution repeats

**file\_categories**

This table stores File Type Categories, which can be created in the WebUI and are used for grouping together multiple file types. They can be used in the Query Editor to select multiple file types at once.

column	type	description
file_category	text	unique identifier and name of a File Type Category
file_types	jsonb	list of all the file types associated with the File Type Category



# MdH-WebUI/Query-Server

## MdH-WebUI/Query-Server

This server is written in Java and offers multiple functionalities. It hosts the WebUI, offers a GraphQL API and other functionalities supporting the WebUI. This component uses [GraphQL-Java](#) to query the PostgreSQL database for file metadata information. Users can directly interact with the API, use a GraphiQL console, or use the Web User Interface to retrieve metadata information.

## Application Programming Interface

The GraphQL queries are sent to the http-server using the http POST method. Port and URI of the server are set by the configuration file (metdatahub/configs/environments.deployment.json). The default Port is 8080, and the default URI is localhost, and the path to the http POST interface is "/graphql".

Using all that information we can now send GraphQL-queries to the server. When the server is running a GraphiQL test console can be found here: <http://localhost:8080/testconsole/> In the console we find information about the query's syntax and what it will retrieve. GraphiQL offers Syntax highlighting and autocompletion and a documentation of our GraphQL Schema.

## GraphQL Schema

Now we will take a closer look at the GraphQL Schema which defines the syntax of our query. Only the most important information is displayed here. The exact schema can be found in /metdatahub/server/src/main/resources/schema.graphqls and the GraphiQL console also visually presents the schema and it's documentation.

The screenshot shows the GraphiQL console interface. On the left, a query is entered: 

```
1 query
2 {
3   searchForFileMetadata
4   (
5     limitFetchingSize: 5,
6     sortBy: ["id"]
7     sortBy_options: [ASC]
8     file_types: ["ARW"],
9     metadata_filter_logic_options: and,
10    selected_attributes: ["FileName", "ExposureTime", "ShutterSpeed"]
11  )
12  {
13    files
14    {
15      dir_path,
16      name,
17      type,
18    }
19    metadata
20    {
21      name,
22      value,
23    }
24  }
25 }
26 }
```

 The right pane shows the JSON response for the query: 

```
{
  "data": {
    "searchForFileMetadata": {
      "files": [
        {
          "dir_path":
            "/home/yupidupidubi/Pictures/testDir/09.19 Ameisen 01",
          "name": "DSC01187.ARW",
          "type": "ARW",
          "metadata": [
            {
              "name": "FileName",
              "value": "DSC01187.ARW"
            },
            {
              "name": "ExposureTime",
              "value": "0.01"
            },
            {
              "name": "ShutterSpeed",
              "value": "0.01"
            }
          ]
        }
      ],
      "dir_path":
        "/home/yupidupidubi/Pictures/testDir/09.19 Ameisen 01",
      "name": "DSC01413.ARW",
      "type": "ARW",
      "metadata": [
        {
          "name": "FileName",
          "value": "DSC01413.ARW"
        },
        {
          "name": "ExposureTime",
          "value": "0.01"
        }
      ]
    }
  }
}
```

 The rightmost pane shows the documentation for the `searchForFileMetadata` query. It includes a description: "searchForFileMetadata: Searches for all file metadata dependent on the specified search options. If no options are specified, all file metadata is returned." It also lists the arguments: `file_ids: [Int!]` (file\_id: Only returns file metadata belonging to one of the file ids in the list), `crawl_ids: [Int!]` (crawl\_id: Only returns file metadata belonging to one of the crawl ids in the list), `dir_path: String` (dir\_path: Only returns file metadata where their directory path matches the specified pattern. Default PatternOption is "included"), `dir_path_option: MetadataOption` (dir\_path\_option: Different PatternOptions can be used, which change how "dir\_path" gets compared to other Strings), `file_name: String` (file\_name: Only returns file metadata where their file\_name matches the specified pattern. Default PatternOption is "included"), `file_name_option: MetadataOption` (file\_name\_option: Different PatternOptions can be used, which change how "file\_name" gets compared to other Strings), and `file_types: [String!]`.

[Video using GraphiQL in the WebUI](#)

### Query: searchForFileMetadata

"searchForFileMetadata" is the only Query of our Schema. It uses a multitude of arguments to filter out the returned file metadata, additionally it can sort the returned files and only return parts of the result set. This makes it possible to query for the whole set of file metadata in multiple queries.

### Object Type: ResultSet

"ResultSet" is the Object Type "searchForFileMetadata" returns. "ResultSet" includes information about all the file metadata, occurred errors and what kind of range of the total query it returned.

### Object Type: File

The Object Type "File" is used for returning information about file metadata. "ResultSet" has a field called "files" which is of the Type File, this is where the file metadata is returned. Most metadata information is stored in its own field "metadata".

## Web User Interface

The server also hosts the WebUI, which offers an easier to use interface for users to send queries but also offers the admin access to the crawler More information about the WebUI can be found on its own documentation page.

## Supporting Services for the WebUI

---

The server also offers services to the WebUI that simplify the query construction. It offers the WebUI file type and metadata attribute information for autocompletion, information about the metadata attribute's datatype, downloading the result set as a json file, storing queries and managing File Type Categories, which group different file types together. More information about the WebUI can be found on its own documentation page.

# WebUI

The WebUI has two main parts, one is the constructing and sending of GraphQL-queries to the server, the other one is starting, stopping and scheduling the crawler.

## Query Editor

The Query Editor is an easy to use interface to create GraphQL queries, which can be send to the server to retrieve file metadata. The Query Editor uses filters to reduce the result set of the returned file metadata.

0100  
10110  
01011

MdH

Metadata-Hub

Query-Editor

Hash-Query

GraphQL

About

Logout

Logged in as **Herbert**  
[enduser]

Toggle Menu

Dark-Mode

Query-Editor

Query-Store

Query Editor

The Query Editor is used for creating queries, which return file metadata.  
Queries are saved in the Query-Store, so they can get executed at a later point in time, without filling in the information again.  
In the query multiple filters can be used to limit the result set of the returned file metadata.

Query-Name [?]  
Default Query Name

Query-Owner [?]  
Herbert

Select File Type Category

Clear selected File Types

File Type Filters: [?]  
ARWJPEG

Open Metadata-Attribut-Selector

Metadata Filters: [?]  
Pattern included ▾ FileNameKandins  
Pattern included ▾ Metadata-AttributeValue

Metadata Filter Connector [?]  
All AND

Show Time Filters

Returned Metadata Attributes: [?]

Show deleted files [?]

Video on how to use the Query Editor

## Query Store

The Query Store is used for saving and restoring queries in the Query Editor.

0100  
10110  
01011

MdH

Metadata-Hub

Query-Editor

Hash-Query

GraphQL

About

Logout

Logged in as **Herbert**  
[enduser]

Toggle Menu

Dark-Mode

Query-Editor

Query-Store

Query Store

The Query Store is used for saving queries, which were already executed or specifically saved.  
These queries can get restored, which will fill in the query information in the Query Editor.

Author-Filter  
Herbert

Herbert	Default Query Name	2020-07-21 17:07	Restore	Delete
Herbert	Default Query Name	2020-07-21 17:07	Restore	Delete
Herbert	Default Query Name	2020-07-21 16:07	Restore	Delete
Herbert	Default Query Name	2020-07-21 16:07	Restore	Delete
Herbert	Default Query Name	2020-07-21 16:07	Restore	Delete
Herbert	Default Query Name	2020-07-21 16:07	Restore	Delete
Herbert	Default Query Name	2020-07-21 16:07	Restore	Delete
Herbert	Default Query Name	2020-07-21 14:07	Restore	Delete
Herbert	Default Query Name	2020-07-21 14:07	Restore	Delete
Herbert	Default Query Name	2020-07-21 14:07	Restore	Delete
Herbert	Default Query Name	2020-07-21 14:07	Restore	Delete
Herbert	Default Query Name	2020-07-21 14:07	Restore	Delete
Herbert	Default Query Name	2020-07-21 14:07	Restore	Delete
Herbert	Default Query Name	2020-07-21 14:07	Restore	Delete
Herbert	Default Query Name	2020-07-21 14:07	Restore	Delete
Herbert	Default Query Name	2020-07-21 14:07	Restore	Delete
Herbert	Default Query Name	2020-07-21 14:07	Restore	Delete
Herbert	Default Query Name	2020-07-21 14:07	Restore	Delete

Video on how to use the Query Store

# File Type Categories

File Type Categories are used to group multiple file types in one category. They can get selected in the Query Editor, to limit the query to specific file types.

Query-Editor

Hash-Query

GraphQL

TreeWalk

About

Logout

Logged in as **Herbert**  
[admin]

Toggle Menu

Dark-Mode

Query-Editor

Query-Store

File-Type-Categories

## File Type Categories

File Type Categories are used to group multiple file types in one category.  
The File Type Categories can get selected in the Query-Editor, to limit the query to specific file types.

All File Type Categories

Create File Category

Update File Category

Delete File Category

File Category [\[?\]](#)

images

Enter a File Category Name

Clear File Types

File Types [\[?\]](#)

ARW

JPEG

PNG

PSD

GIF

[Video on how to use the File Type Categories](#)

# Hash Query

The Hash Query is only used to check if a file is already in the database. It looks for the sha256-hash of the file in the database, the hash can be manually typed or calculated from a file on disk.

# GraphiQL-Console

The GraphiQL Console is integrated into the WebUI, it offers syntax highlighting, corrections and autocomplete for GraphQL queries depending on the GraphQL Schema. The GraphQL Schema documentation can also be looked at using the console. More information about GraphQL and GraphiQL can be found in the [server-section](#).

# TreeWalk Controller

The TreeWalk Controller is used for controlling the TreeWalk and its executions.

Query-Editor

Hash-Query

GraphQL

TreeWalk

About

Logout

Logged in as **User**  
[admin]

Toggle Menu

Dark-Mode

Controller

Scheduler

Intervals

## Controller

This is the panel for controlling the TreeWalk. The TreeWalk has three states: **ready**, **running** and **paused**. All these actions are safe to use in all states, but some may have no impact in a certain state. For example, stopping the TreeWalk when it was running will stop the current execution, but stopping when the TreeWalk was ready will have no consequences. Make sure to wait for the response if you invoked an action. You'll see an alert message at the bottom once the action has finished. Especially starting the TreeWalk might take some time due to the generation of the work packages.

READY

## Actions

Shutdown the TreeWalk entirely. Stop a possible current execution and terminate all TreeWalk threads.

SHUTDOWN

Start a TreeWalk execution. Shows the configuration panel for a manual insertion of the configuration data.

START

Message Configuration with identifier 8de10680cc9bf8589458433d246dec14d9778f373fc869ffca5ee71cctfbab1cd was successfully added to the schedule.


Info Time: 7/22/2020, 3:16:12 AM , Command: add-config

[Video on how to configure a TreeWalk execution.](#)

[Videon on how to control the TreeWalk](#)

# TreeWalk Scheduler

The TreeWalk Scheduler is used for getting an overview about scheduled TreeWalk executions and removing them from the schedule.



[Query-Editor](#)

[Hash-Query](#)

[GraphQL](#)

[TreeWalk](#)

[About](#)

[Logout](#)

Logged in as **User**  
[admin]

[Toggle Menu](#)

☐ Dark-Mode

Controller **Scheduler** Intervals

## Schedule

crawlHome

WAITING PERIODIC

User

The next execution is scheduled at: 2020-07-29 03:15:00

Repeats every: 1 days 0 hours 0 minutes 0 seconds

CONFIG REMOVE

crawlPictures

WAITING FORCE PERIODIC

User


The next execution is scheduled at: 2020-07-30 03:15:00

Repeats every: 1 days 0 hours 0 minutes 0 seconds

CONFIG REMOVE

## TreeWalk Intervals

Intervals are used to create intervals, which can change the maximum resource consumption of the crawler during certain time intervals in a week.



[Query-Editor](#)

[Hash-Query](#)

[GraphQL](#)

[TreeWalk](#)

[About](#)

[Logout](#)

Logged in as **Herbert**  
[admin]

[Toggle Menu](#)

☐ Dark-Mode

Controller Scheduler **Intervals**

## Intervals

This is the configuration panel for the administration of time intervals for maximum resource consumption. These time intervals can be added and deleted using the form and the remove buttons on each item. They are periodically defined for each week. The page is refreshed each minute or upon adding/deleting time intervals. If you want to update it manually, just click the refresh button.

Start

Monday

17

0

End

Tuesday

7

0

CPU Level

3

SUBMIT

CLEAR

Refresh

[Video on how to configure an interval](#)

[Video showing the effect of active time intervals on the executions of the TreeWalk](#)

# Installation

This chapter will show you how to install the software requirements and the Metadata-Hub application. It is important to mention that this is **no** application that should be used in a production environment because of predefined user/password settings that cannot be changed.

## Requirements

The application is published as a Docker image. Thus, it requires your system to have the Docker Engine installed. Therefore, please refer to the official installation instructions of Docker at <https://docs.docker.com/get-docker>. Please make sure to install at least version *19.03* and check the installation before continuing.

## Usage

This guide is written based on a working Docker setup on Linux. If you want to use the Software on Windows, please follow the procedure but modify the commands according to your setup. When you're using PowerShell on Windows, you can use the same commands as described below.

### Pulling the image

In the first step, pull the image from DockerHub. The `latest` version is the latest stable version that is updated on each release. The `dev` version is the currently developed version and may still contain errors. Execute the following command in order to pull the latest stable image.

```
$ docker pull amosproject2/metatahub:latest
```

### Persisting the database

If you want to persist the database on your machine, a Docker volume is required. Mounting an arbitrary directory will lead to a failure during the PostgreSQL startup. Here is an example that shows how to create a local volume using the Docker CLI. For more information, please have a look at the [official documentation](#).

```
docker volume create --name metatahub-database -d local
```

### Running Metadata-Hub

The image can be started according to the following command:

```
docker run \
  -p {ui-port}:8080 \
  -p {treewalk-port}:9000 \
  -p {database-port}:5432 \
  -v {data}:/filesystem \
  -v {volume-name}:/var/lib/postgresql/12/main \
  amosproject2/metatahub
```

- `ui-port`  
The port that publishes the web interface on the *host* machine.
- `treewalk-port`  
The port that publishes the TreeWalk API on the *host* machine. Setting this port is only required when you directly want to access the TreeWalk API and can be omitted.
- `database-port`  
The port that publishes the database instance on the *host* machine. Setting this port is only required when you directly want to access the database and can be omitted.
- `data`  
This directory will be mounted inside the container and therefore is accessible for the TreeWalk.
- `volume-name`  
The name of the volume that should be used to persist the database on the host machine over multiple runs. In the example from above, the volume name would be set to `metatahub-database`

The internal ports `8080`, `9000` and `5432` must **not** change because they are required for internal communication. Of course, multiple directories can be mounted inside the container, simply provide each directory with the corresponding `-v` flag.

This example starts a container with access to the `/home/data` directory.

```
docker run \
  -p 8080:8080 \
  -v /home/data:/filesystem \
  -v metadatahub-database:/var/lib/postgresql/12/main \
  amosproject2/metadatahub
```

Please refer to the [Usage Guide](#) for a demo of how to use the application. Make sure to provide the filepaths relative to mounted directory inside the container.

## Inspecting a container

In order to inspect a container with the ID `container-id`, start a bash session inside the container.

```
docker exec -it {container-id} /bin/bash
```

The log files for further investigation are directly located in the `/metadatahub` root directory.

If you encounter any errors, please refer to the [FAQ](#) section.

# FAQ

## I have problems with docker permissions using Linux.

It is most likely your user does not belong to the `docker` group. Please have a look at these [instructions](#).

## I have problems with a running container and want to inspect them.

You can start a shell session in your running container:

```
docker exec -it {container-id} /bin/bash
```

This will start a *bash* session inside the container with the ID `container-id`. Furthermore, you can inspect the log files at `/metadatahub/server.log` and `/metadatahub/crawler.log`.

## I have problems with scheduled exeutions and time intervals

The docker image uses UTC time instead of the local time on your system. All time values, such as the start of a TreeWalk exeuction, should be given in UTC time.