Improvement of the Rucio implementation for the LDCS platform and search for dark data.

Department of Particle Physics

Bachelor report

Piotr Yartsev

Spring, 2021

Project duration:
Full time
4 months

Supervisors:
Ruth Pöttgen
Balazs Konya

Abstract

In this work we aim to implement a software package to detect and categorize dark data, data not accessible or not known by the user, generated in the simulations of the Light Dark Matter eXperiment (LDMX). This will involve studying current existing solutions for such problems, attempting to implement them for the Lightweight Distributed Computing System (LDCS), and developing our own Dark Data Search (DDS) toolkit to perform the detection and categorization of the dark data. The result provided by these tools will be examined further for clues as to why and how dark data was created. Physics simulations of the LDMX detector were executed to create dark data, allowing us to study the conditions for their creation, and to get a deeper understanding of the physics for a missing momentum dark matter experiment. Based on the research done in this paper a multitude of systematic problems was found that would require addressing for the LDCS.

Contents

1 Introduction **1**

2 What is LDMX, LDCS and Rucio? **2**
2.1 Physics background and the LDMX detector . . . 2
2.2 LDCS . . . 4
2.3 LDCS simulation workflow . . . 6
2.4 Rucio . . . 7
2.5 How does Rucio work? . . . 8
2.6 What are some problems with the current implementation? . . . 9
2.6.1 Data-metadata discrepancies . . . 9
2.6.2 Reason for discrepancies . . . 9
2.7 Proposal for new dark data search toolkit . . . 10

3 Dark Data Search toolkit **12**
3.1 Comparison tool . . . 12
3.1.1 Initializing the comparison tool . . . 12
3.1.2 Running the comparison tool . . . 14
3.2 Classification tool . . . 16
3.2.1 Retrieve information about the output of comparison tool . . . 16
3.2.2 Running the tools . . . 16
3.2.3 Classification of files missing from storage . . . 16
3.2.4 Classification of files missing from Rucio . . . 17

4 Dark data search and analysis at LDCS **18**
4.1 Running our own simulations campaign: trying to understand dark data generation . . . 18
4.1.1 Small scale simulation campaign . . . 18
4.1.2 Dark data appearance during reconstruction . . . 18
4.2 Search and analysis of generated dark data at LDCS storages . . . 19
4.3 LDCS shortcomings discovered during the dark data search . . . 23

5 Results **24**

6 Outlook **25**
6.1 Missing features and bugs. . . . 25
6.2 Future work . . . 26

7 Acknowledgements **27**

8 Figures and tables **28**

9 References **36**

10 Code **37**

1 Introduction

The Light Dark Matter eXperiment (LDMX) [1] is a proposed accelerator-based experiment to search for light dark matter particles. The origin and observed abundance of Dark Matter in the Universe can be explained elegantly by the thermal freeze-out mechanism, leading to a preferred mass range of the Dark Matter particles in the MeV-TeV region [1, 2]. The GeV-TeV mass range is being explored intensely by a variety of experiments searching for Weakly Interacting Massive Particles. The sub-GeV region, however, in which the masses of most of the building blocks of stable matter lie, is hardly being tested experimentally to date. The Light Dark Matter eXperiment (LDMX) is a planned electron-beam fixed-target experiment, that has the unique potential to conclusively test models for such light Dark Matter in the MeV to GeV range. The proposed electron beam for this experiment is a 4 to 8 GeV low current offshoot from the Stanford Linear Accelerator Center (SLAC) DASEL beamline [3] in San Francisco. By measuring each incoming electron we can compare the incoming transverse momentum of the electron, as measured by the tagging tracker detector 2.1, with the outgoing momentum, as measured by the recoil tracker 2.1, too see if there is a loss of total transverse momentum. If no particles are detected in the other detectors at LDMX that can explain that difference in momentum, we can attribute it to a Dark Matter particle.

At this stage, LDMX is undergoing feasibility and design studies which require massive computer simulations and complex infrastructure to handle, analyze and store all the data generated. To do that the Lightweight Distributed Computing System (LDCS) [4] was created and it is using a range of different hardware and software, including the Rucio [5] distributed data management system designed for the ATLAS experiment [6] at CERN. In the LDCS platform, the Rucio system is used to catalog the data from all the storage sites, such as the data stored locally at the simulation site and the data stored at the primary storage LCSC at SLAC, generated by the simulations.

Currently, the information stored by Rucio does not always match the data kept in storage, meaning that we have dark data. Dark data is a term that describes data that exist in storage but the system or users will never access. This includes data files that were put in storage on purpose to be used but never were or a situation where the user/system is not aware of the data's existence, which is the problem in our case. Removing this dark data or reintroducing it to the user so it can be used again is important, but this dark data has to be discovered before we can do that. Discovering, analyzing it to find its origin, and replicating the conditions that lead to the dark data is the main problem this bachelor's project aims to solve. Recreating this dark data would require running the simulations of the LDMX detector, so an understanding of the physics of such an experiment and how that sort of physics is simulated is of great importance.

2 What is LDMX, LDCS and Rucio?

2.1 Physics background and the LDMX detector

Dark matter was first theorized as an explanation for the mismatch between the theoretically predicted rotational speed of galaxies and the measured result [13]. To correct for this a type of matter was predicted that did not send out any measurable light, hence the name Dark Matter, but interacted with the visible Standard model (SM) particles in the galaxy through the gravitational force. In the following decade, several other cosmic phenomena indicated that the models that only accounted for the visible matter were not sufficient to describe those measurements. One theory for the origin of Dark Matter (DM) is the thermal freeze-out mechanic [2] which states that dark matter existed in thermal equilibrium with Standard Model (SM) particles in the hot early universe. This theory says that the annihilation rate of DM into SM, which can be written as

$$n_X \sigma v$$

where n_X is the number density of DM and σv is the velocity averaged cross-section of the interaction between DM and SM, fell below the Hubble expansion rate during the early universe, leading to a stable abundance of Dark Matter in the universe. This theory allows us, using the known quantity of dark matter in the universe now and models for the expansion of the early universe, to calculate the thermal average cross-section to be

$$< \sigma v > \simeq 3 \cdot 10^{-26}\ cm^3/s$$

which sets a limit on the possible mass of DM particles to be in the 10 keV $-$ 100 TeV region. There exist many other theories that attempt to explain the creation of dark matter that also fall in the same region for possible DM masses, making it an interesting region to probe.

The LDMX experiment will focus its research on the sub-region of MeV to GeV where it will attempt to detect DM using a missing momentum experiment, which relies on the law of conservation of momentum. If we know the momentum of the incoming particle and can measure the moment of some of the outcoming particles, we can calculate the momentum of any outcoming particle that was not measured by the detector. This makes this method very attractive for a search for DM particles, as we do not have to detect them directly but only have to measure the SM particle's output. The missing momentum detector uses the curvature of the particle in the detector to calculate the momentum of the particle using

$$p \propto Bqr \tag{1}$$

where B is the magnetic field, q is the charge of the particle and r is the radius for the curvature of the particle traveling through the magnetic field.

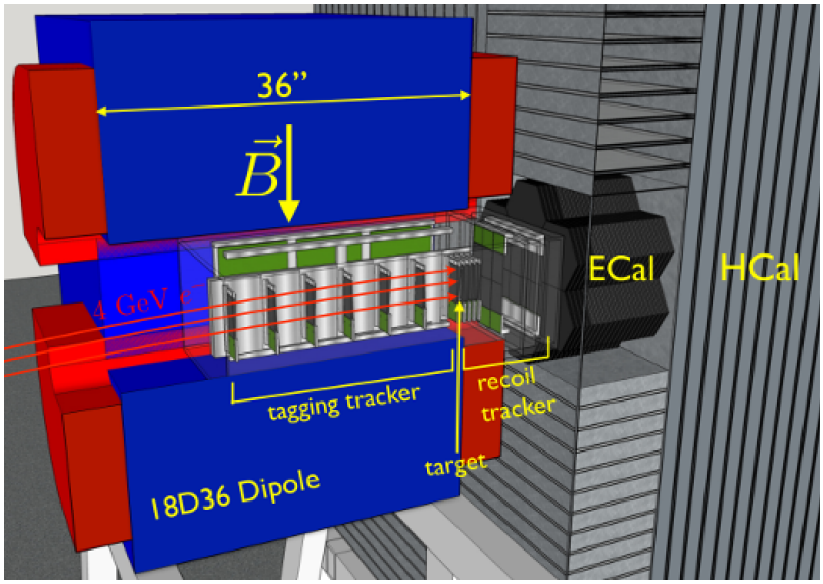


Figure 1: A overview [16] of the different parts making LDMX detectors with the tagging tracker, the dipole magnet surrounding it, the target followed by the recoil tracker, electronic calorimiter (ECal) and finally the hadronic calorimiter (HCal).

LDMX will consist of four main detectors, as can be seen in Figure 1:

The tagging tracker

This detector is responsible for determining the momentum of the incoming electrons. While the electron beam is designed to run at particular set energy, for example 4 GeV, individual electrons in the beam may stray from that value. These electrons have to be discarded in our measurements and we can use the momentum measurement in the tagging tracker for that task. This detector also allows for precise measurement of the momentum of the individual electrons passing through it, which is something we would need in a missing momentum experiment. Using the effect shown in equation 1 the tagging tracker may use the surrounding dipole magnet to alter the path of the electron. The individual tracking detectors are placed at an interval between them along the path of the beam, allowing us to measure this deviation from the straight path of the electron and therefore with high precision determine its momentum.

The recoil tracker

The recoil tracker functions in a similar fashion as the tagging tracker detector using the effect shown in equation 1 and using the fringe field generated by the dipole magnet mentioned in 2.1 to determine the momentum of electrons passing through it. The difference from the tagging tracker detector is the purpose of it is to measure the momentum of electrons recoiling against the target. This means that it is designed to also be able to detect electrons with lower energy and momentum than the ones we see in the beams as well as covering a wider angle to detect all the recoiling electrons and other charged particles.

Electromagnetic calorimeter (ECal)

The purpose of the ECal detector is to measure the energy of the electrons that pass through the target. Since any reaction with the stationary target by the electron that could produce DM requires loss of energy for the electron we can use a trigger system where only events with lower recorded energy in the ECal than the beam energy are considered candidate events. The ECal can also be used for the identification of charged hadrons, while the task of identifying the neutral hadrons is left to the HCal.

The Hadronic Calorimeter (HCal)

The main purpose of the hadronic calorimeter is to detect possible neutral hadronic products of electrons reaction with the target or with the ECal detector.

Currently, the project is in its planning state so we need to generate simulation data to use in the meantime. For this task LDMX project uses the Geant4 [14], a simulation toolkit that allows us to simulate the interaction between particles and matter, such as the detectors. Geant4 was developed at CERN and has been used by many particle physics experiments, such as ALICE and ATLAS at the LHC, and has gone through multiple calibrations where simulated data was compared to measured results. It can therefore be reasonably assumed to be a good tool for predicting and modeling a particle physics experiment, such as LDMX.

There are two main purposes for the Geant4 software package at LDMX/LDCS:

I. To model the background that LDMX would measure in its detectors, which is vital for an experiment to be able to detect events beyond what is predicted to arise from background processes.

II. To create simulated data similar to what we would expect from the actual experiment, allowing us to verify that the setup planned for the actual experiment is functional. This is particularly important for us at LDCS so that we know that all the digital infrastructure operates correctly with the type of data we would expect from the experiment.

As with any particle physics experiment, LDMX will detect many background processes that have to be accounted for to be able to state a discovery of DM particles. One such background process is when a photon undergoes a photo-nuclear reaction, which is something we will simulate as part of the project and is explained further in the next section. These photons are produced by hard bremsstrahlung, which is when the deceleration of a charged particle, in or case the electrons from the electron beam, produces a photon. These photons can then undergo a photonuclear reaction where the photon is converted to hadrons in the target and sometimes inside the ECal. This results in the electron losing energy to the photon, meaning it will pass the energy trigger, and has a signature that is very difficult to differentiate from a DM event. If we are unable to detect the photon or its byproducts, this event might appear as a loss of the total momentum, which might be interpreted as a DM event.

As stated in the section regarding the LDCS simulation workflow 2.3, to run a simulation we need to create a configuration file that details what physics process we want to study. This includes specifying variables such as the electron beam energy, what detectors we want to measure the results from, what version of the detector setup we are simulating, and other similar physics parameters. The output from such a simulation is a root file, a format that is readable by the ROOT data management and plotting tool developed by CERN [15], which is a standard tool used by many particle physics experiments.

2.2 LDCS

For small-scale particle physics experiments, such as the LDMX, it is very challenging to design and upkeep e-infrastructure due to limits on the available resources. In 2020 it was decided to create a prototype e-infrastructure system for running simulation tasks, called Lightweight Distributed Computing System (LDCS). Due to the aforementioned difficulties in creating and maintaining such infrastructure for a small team of researchers, the system was designed to integrate existing solutions to the greatest extent that is possible to reduce the need of developing our own software. The LDCS, just as the name implies, uses distributed computing system for running simulations with locations in California, USA, and Lund, Sweden.

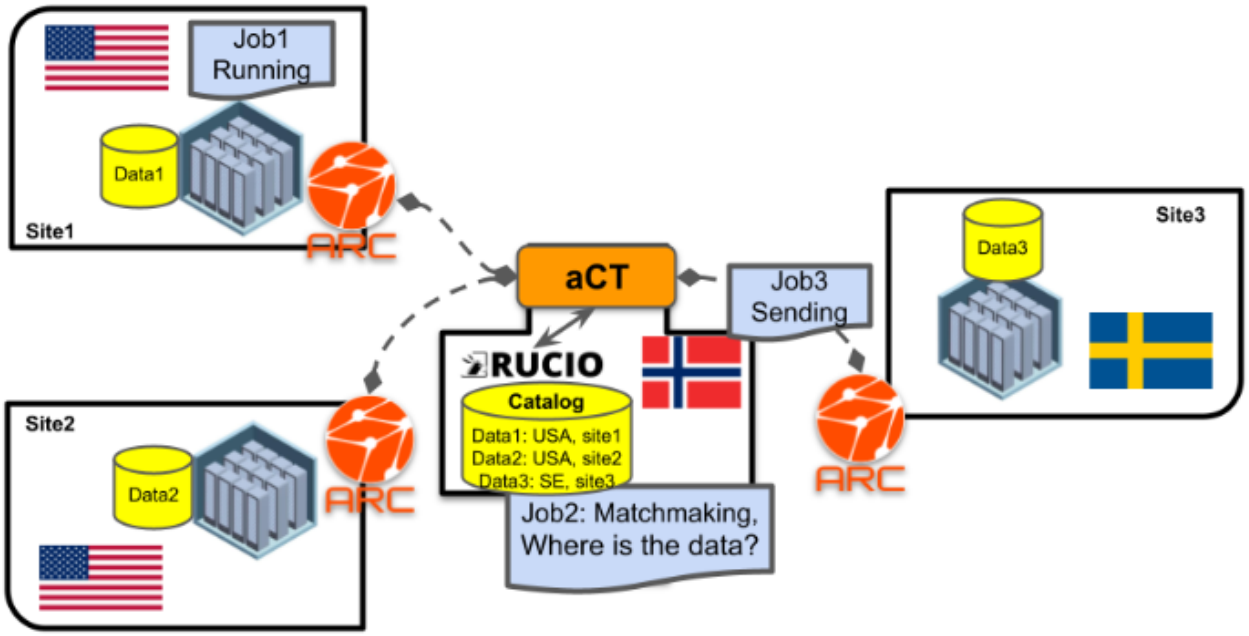


Figure 2: Thee setup for the LDCS with three computational sites, each with an installation ARC receiving job submissions from ACT with the Rucio software cataloging all the generated data. [4]

The LDCS consists of several core components which can be seen in Figure 2:

Advanced Resource Connector (ARC)

ARC acts as a middleware between the computing centers and the job submission system. Its main component is the ARC Resource-coupled EXecution service (A-REX) which performs authentication of LDCS users and allows the execution of simulation jobs. The A-REX system is also responsible for uploading the output from the simulation job to the correct storage location.

ARC Control Tower (aCT)

This service allows the submission of simulation jobs to the different computational cites. For LDMX we use several configuration files, which are further explained in section 2.3, to define the simulation parameters and a custom LDMX-system interprets those files and submits a job request to the aCT. aCT then determines based on the number of queued jobs on each computational site where to send the job for simulation.

Rucio

Rucio is a data cataloging software that is used to keep track of where files are located. More information regarding Rucio can be found in section 2.4.

LDMX Software

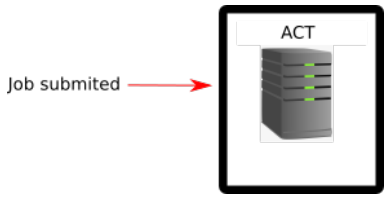
LDMX Software is a modified version of the detector simulation application Geant4, which you can read more about in section 2.1. This software is installed on all computational sites and is used to execute the simulation jobs based on the parameters it receives from ARC.

Monitoring the System

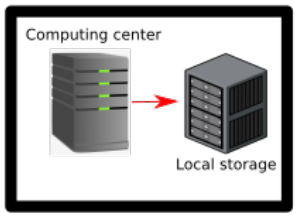
To track the state of the LDCS, a graphical web-based dashboard was created that displays information such as the number of queued or running jobs, storage use, number of failed jobs, etc. It uses the open-source visualization tool Grafana to display the aforementioned information about the state of the system using time-series graphs.

2.3 LDCS simulation workflow

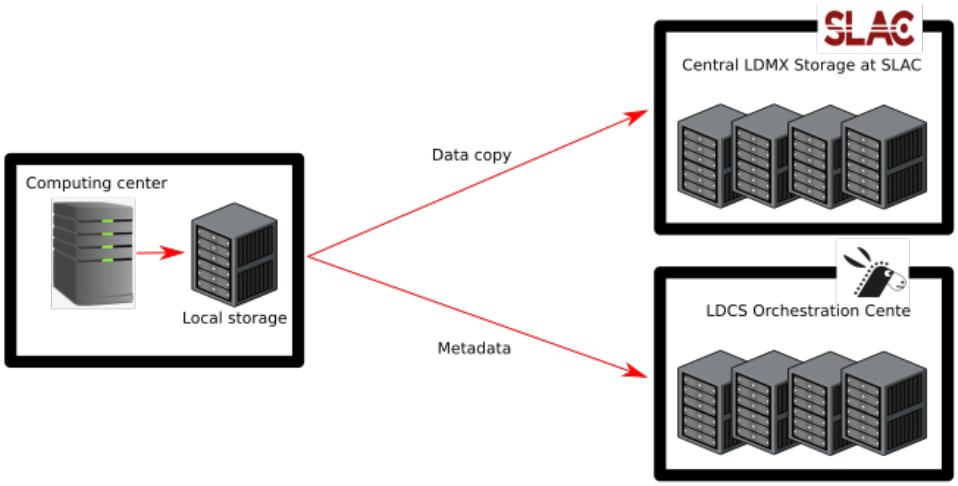
A batch of simulation jobs is submitted to ARC Control Tower (ACT) with a configuration file that details what simulation parameters the user wants to use for this simulation. The ACT then distributes the workload among the computation centers.



The computation center runs the simulation and the generated output file is stored locally at the computing center. Post-processing software is run to generate a persistent file name, calculate the checksum, and other similar operations. This information as well as information from the simulation parameters are written to a metadata file. See Figure 9 for an example of a metadata file.



A copy of the simulation output data is sent to the Central LDMX Storage (CLS) at SLAC. The metadata and information about the location for the output data file is registered in the Rucio cataloguing software running at the Rucio server in Oslo, Norway.



2.4 Rucio

Rucio [5] is open-source software written in Python for organizing and managing a large amount of data, possibly split among many different storage locations. It was developed for the ATLAS experiment at LHC [6] to meet the requirement of managing petabytes of data generated by the experiment. It has since its development been used in many other experiments, including the LDCS platform [1]. While Rucio is an advanced software containing a multitude of tools, it is only used for cataloging data. Rucio does not store the data in any capacity, only metadata about the data. The actual data can be stored at many different locations with different storage capacities simultaneously and Rucio will be able to keep track of what file is located where, but it will never store any of those files itself. Using Rucio it is possible to initialize many data administrative tasks, such as transfer or deletion of files, but that action is not performed by Rucio itself. It relies on existing file transfer frameworks and protocols to do that job and only informs the storage site about where to move what file using what framework.

In the LDCS platform, Rucio is responsible for cataloging data generated by the simulations of the detector planned to be constructed at SLAC in San Francisco. LDCS is expected to handle data in the order of 10 Petabytes [4] generated by the distributed computing system with three computational centers located in California, USA, and one in Lund, Sweden. The data from the simulation is stored locally at the computational centers as well as a copy at the primary data storage hub at SLAC in San Francisco, USA. The LDCS implementation of Rucio was designed to be lightweight, therefore the minimum setup was used meaning that not all parts of the Rucio software are implemented. The Rucio software allows for many operations related to the catalog and allows for a search for entries using several filters. These filters include searching by storage location, also known as Rucio Storage Elements (RSE), searching by scopes, which is a pre-grouped collection of entries, and searching by Data IDentifier (DID) which allows us to search for entries with specific parameters, for instance only search for files submitted by a specific user. The current instance of the Rucio software is running on a server in Oslo, Norway, and at the moment of writing (02/05/2022) about 1 PB of storage has been filled with simulation data and every one of these files has to be accounted for.

2.5 How does Rucio work?

Rucio software consists of 4 parts (see Figure 3).

- Client
- Core/Server
- Daemons
- Analytics (Not relevant for this thesis)

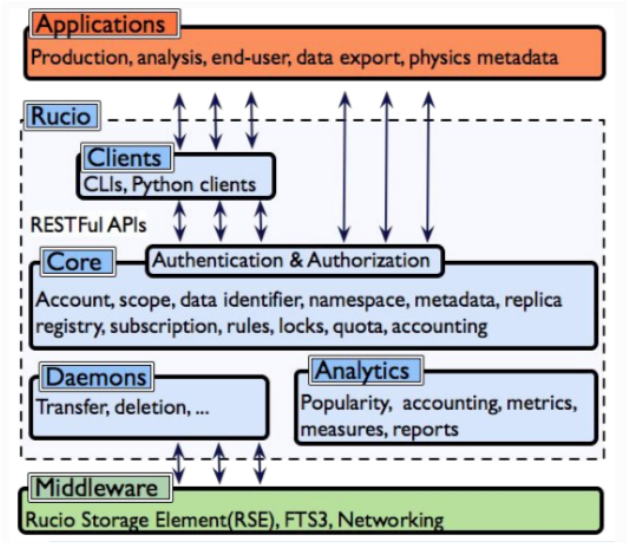


Figure 3: The Rucio system is based on a distributed architecture and can be decomposed into four main components: Client, Core, Daemons and Analytics [10].

The client part of the software is designed to allow researchers (clients) to interact with the Rucio software and execute commands from clients' own devices. Accessing the Rucio system is commonly done through the Command Line Interface (CLI), but can also be accessed using the Python clients and the JavaScript-based web user interface. Using the CLI interface allows you to execute commands in a computer terminal and common commands can be found in the Rucio documentation [7]. This requires the user to install the Rucio client on a computer. Rucio has an API that simplifies the usage of the REST interface in Python by implementing an authentication token and a Python wrapper for most of the REST interface commands. The documentation for the API can be found on references [8, 9]. Although the documentation is at times lacking in details, it seems to allow most of the actions possible using the CLI. My bachelor project is intended to be written, to the largest extent possible, in Python so familiarising myself with these tools is vital.

The server part refers to the part of the software that is running on the server hosting Rucio which is listening for incoming queries. It contains several Rucio core components, such as account authorization, communicating with storage, data cataloging, and metadata. It allows the client to access many different tools using a single interface, reducing the complexity to perform querying operations in Rucio. The Core part is responsible for interacting with the data storage. Rucio will connect with the data storage in a way that is compatible with the storage's system and protocol and will then use the File Transfer System (FTS) to transfer the data between the storage locations.

The daemon part contains the software responsible for executing continuously running background processes. Rucio has many daemons that perform many possible actions, but they are not mandatory and do not have to be run if they are deemed not necessary for the operation of the whole system. An example of a Rucio daemon is the rucio-dumper, which compiles a list of the files currently present at the storage location, and the rucio-auditor daemon, which compares the information registered in the Rucio catalog with the list of data provided by the rucio-dumper to find inconsistencies. Both of these tools are related to the search for dark data and can be useful in my bachelor's project.

2.6 What are some problems with the current implementation?

2.6.1 Data-metadata discrepancies

Currently, there does not exist a method for comparing and syncing up the Rucio catalog with the data, so discrepancies can arise. Comparing the Rucio catalog with the data files would currently have to be done manually, and an automatic solution needs to be implemented. The issue of the duplicate files was detected by one of the researchers working on the LDCS while looking at the output of simulation jobs. After this issue was reported to the rest of the LDCS development team a simple program was written that counted the number of files in storage and the number registered in the Rucio catalog and found a discrepancy.

One possible solution for detecting these discrepancies is to use tools already provided by the Rucio software. As mentioned in section 2.5 there exist two daemons that could be used in combination to solve this problem: rucio-dumper and rucio-auditor. Rucio-dumper creates a list of the directories containing the data in storage and writes the locations to a .txt file. The rucio-auditor retrieves all the files in the directories described in the dump .txt file and searches for them in the Rucio catalog.

Another possible solution is designing similar tools to the two daemons mentioned above myself. This would not be too difficult and can be performed using a Python and/or Bashscript. As long as the device running the script has access to both the data and the Rucio catalog it is possible to compare the two. A disadvantage with such a solution is that it would most likely be less efficient than what was created by the Rucio team due to them having access to more development resources and access to many experienced developers at CERN, the developers of Rucio. An argument for developing our solution is that the efficiency is less important if the difference between the efficient and inefficient solution is minuscule compared to the workload the data storage and the Rucio database experience normally. This is difficult to know without testing both solutions and knowing the amount of computing power and bandwidth we have to work with.

2.6.2 Reason for discrepancies

The solution proposed above does not say anything about what files are causing this problem, only that this dark data exist. It would be preferable to prevent the generation of dark data in the first place. This would eliminate the need to search for dark data in the storage's and in the Rucio catalog as well as saving computational time that was spent generating these dark data files. Therefore, finding the source of this dark data is crucial.

Another reason to find the origin of this dark data is that it might show errors/problems with the current LDCS setup. If we know that the simulation, transfer of files, or the Rucio registration generates dark files that might be an indication of a deeper problem with that part of the LDCS setup. Solving such a problem is not in the scope of this bachelor thesis, but the data generated by the software proposed in this bachelor thesis might be used in the future to solve this problem.

This method relies on the fact that the reason for the dark data can be gathered from the information about the files or metadata entries we have available. If this dark data is generated due to a problem related to something outside the creation of the file/metadata, for instance, it disappearing during transfer to the Rucio server, we would have no way of knowing this using the suggested method. Then further analysis would have to be made, such as looking at what files were sent compared to what files were received. It is also possible that these dark data issues are completely random and have to do with factors out of our control, in which case a statistical analysis would result in no significant correlation between the statistics for the data files and the statistics for the dark data. While this would be unfortunate, it would still be useful information that can allow us to exclude a source for the dark data.

2.7 Proposal for new dark data search toolkit

The original plan was to use both the tools provided by Rucio, such as the auditor-daemon and the dump-daemon, in combination with tools we created on our own to solve our problem with dark data. The advantage of such a solution is that we would get support and updates from the Rucio development team for the tools that they provide, as well as allow us to create tools to solve problems specific to our setup for the LDCS, problems that are too specific for a general solution from the Rucio software package. One combined solution that was discussed was to use the Rucio daemons for the full comparison of the Rucio catalog with the storage and use our tool for a simpler and faster but less thorough comparison.

During the process of evaluating Rucio features, we found several shortcomings that made implementing those features at LDCS problematic. Therefore, we decided not to implement the solution existing as part of the Rucio framework, but develop our complete solution toolkit for the dark data problem. Below we list some of the problems we encountered during our attempt to implement the Rucio features and in section 3 we describe the new toolkit we developed.

- The documentation for the tools provided by Rucio that we planned to use as part of our solution severely lacked details about how the daemons operate and only brief descriptions are given. The documentation for all the different tools in Rucio [10] usually consists of three parts:

  I. The name of the tool

  II. A one to the two-sentence description for the tools function.

  III. A description of the arguments and what arguments can be provided to the tool, be it a CLI command, daemon, or the Rucio PythonRest API.

  An example of such a description is the text explaining the function of the auditor daemon which read as follows: "The auditor daemon is the one responsible for the detection of inconsistencies on storage, i.e.: dark data discovery." [11]. This is the only description we could find of what the tool does and it does not provide any information on how it performs this detection of the inconsistencies, what is the output from this tool, or any specific information about how the tool operates. The Rucio developers were contacted to get a better picture of how the tools operated, but the answers they gave were fairly vague due to the inexperience of individual developers with the whole system. This substation limited the speed of our progress due to us having to spend a significant amount of time researching the daemons by studying the code, looking for examples of implementations online as well as learning through trial and error.

- Another problem was discovered during the research of how the daemons function and it is that the daemon has to run close to the Rucio server, meaning on the same server or a device with direct access to the server, as opposed to being able to run from any device or location. This meant that the safety of such implementation had to be considered much more thoroughly because if it causes some problem that could damage the Rucio server. The Rucio server is an integral part of the LDCS and without it, the whole system would not work correctly, hence we had to make sure the daemons would behave safely and predictably. While I was able to run the daemons in a Docker [12] container, it was unclear how much time it would take to fully study this implementation.

- The last argument against a combined solution is that due to the need for the Rucio daemons to run on the Rucio server I would need access to the server for testing the implementation during development, but I was only granted access to the Lund storage and not the Rucio server. This meant that even if the Rucio daemon implementation was deemed to be safe and was added to the Rucio server, any modification to this implementation would have to go through an admin for the Rucio server, and the back and forth communication would limit the speed at which we can test the implementation. Developing our tools would allow us to constantly tests it on the storage in Lund and even if we would not know if it would work at other storage locations, we would at least have a functional tool for Lund that could be adapted to the other locations

While creating our tools several issues were found with the Rucio Python RESTful API:

- Just as with the Rucio daemon, the documentation for the Rucio RESTful API was lacking in specificity and only included the name of the command, a one-sentence description of what the command does, and a description of what the different arguments for the function meant. This meant a significant amount of time would need to be spent trying to get the commands to work through trial and error.
- It was discovered there exists an inconsistency in the way the Rucio RESTful API handles the argument for its filters. One example of such an inconsistency is where some functions expected a colon sign (:) between the argument for the filter and other functions expected an equal sign (=) for the same filter. This meant that additional time had to be spent trying all the possible visions of an input the functions could expect, which was taking up development time.
- Another problem with the implementation of the Rucio RESTful API is that it requires certain files to be placed in the /lib directory which contains essential shared libraries for a Linux operating system. The files are necessary for the API so it can communicate with the Rucio-client installation and contain information such as the login credentials for the Rucio client as well as a certificate to prove the user. This complicates the installation process for the tools and creates possibilities for incorrect installations. By accessing the Rucio system using CLI commands we only require that it runs in the rucio-client virtual environment, something everyone on the LDCS development team already has access to, so no additional installations or modifications to the system have to be made.

With these issues in mind, the decision was made not to use the Rucio Python RESTful API for interacting with the Rucio server, but instead, to nest Bash commands inside the python code to execute the Rucio CLI commands. Examples of these nested Bash commands can be seen in the code for the comparison tool 2 and in the next section 3 where we describe the new toolkit that we developed.

3 Dark Data Search toolkit

The Dark Data Search (DDS) toolkit was designed to perform two functions at the LDCS:

I. Detection of the different types of dark data

II. Classification of the dark data files to find systematic issues related to the LDCS setup and to gather additional statistics.

The first function is performed by the comparison tool 3.1, which compares the data found in the Rucio catalog with the files present in the storage, to find any mismatches between them. The tool can be configured to run a limited test that only looks for the dark data files or be configured to also detect any corrupt files and improper or missing data locations. The second function is performed by the classification tool 3.2, which uses the output from the comparison tool to find patterns in the dark data. It does it by gathering statistics about the whole system, for example, the number of files per dataset, and uses it to find and flag suspicious files, storage locations, datasets, etc.

3.1 Comparison tool

3.1.1 Initializing the comparison tool

The tool for comparing Rucio with storage was set up to be able to perform many different types of searches so the code can be set up to run in different modes. These functions also make the code more future-proof as even if scopes and RSEs are added or removed from Rucio the code will still know the current state of the system.

```
python DDS_comparison_tool.py rse= scopes= output= limit= checksum=
```

-rse

The argument RSE allows you to search all the storage locations or limit the search to only a limited set of RSEs. This setting serves two purposes: Currently, the code can only function if it has direct access to the storage, for instance running on the machine where the data files are located, so searching for files at a different storage location is not possible and therefore it is advisable for the user to only search for files at the storage location connected to the device from which the tool is run.

If the function for searching at other storage locations is added to the tool, the functions to run the comparison for only a subset of the storage locations might allow us to perform diagnostics of several storage locations at once.

The argument input by the user is checked for being a valid input by requesting a list of all the valid RSE from Rucio and checking if the argument RSE is among the valid ones. There exist an input option “All” which will run the code for all the valid RSE.

-scopes

The argument scopes allow you to limit the comparison tool to only use files registered to a particular scope for the comparison. This is done similarly to the RSE argument: by getting the valid scopes in Rucio and checking if the argument is among the valid input. Similarly, the argument “All” can be used to designate that the tool should run for all the scopes. The function to limit the scopes can be used to study the state of particular scope for consistency.

-output

The comparison tool has the option to display information during the running of the code, such as print out statistics about the different steps of the process, like the number of files found in the Rucio catalog for a specific RSE, as well as display a status bar that informs the user of how what percentage

of a process is complete and predicts how much of it is left. This information can be used to quickly assess the state of the system without the need for the user to study the output files. In case the user does not want to see this information the output argument can be set to “False”, in which case the program will not display this information in the terminal.

-checksum

When running the tool a large amount of time is spent on calculating and comparing the checksum for the files in storage, which is done to check if any files have been corrupted, so choosing if you want to calculate the checksum will greatly impact the time needed for the tool to complete its comparison. Therefore, the setting was created that allows the user to not perform the checksum for the file in storage and only check if it exists there or not. This can be done by setting the checksum argument to be equal to “False”

-limit

To complete a run of the comparison for even a single scope can take anywhere between 10 min to many hours. Therefore the setting was created to set a limit on how many datasets the user wants to check for inconsistencies. This allows the tool to complete a run of the comparison tool much quicker, sometimes in under a minute, while still performing all the steps in the comparison process which allows checking if all the tools a functioning correctly. The limit can be set to any integer number, which will limit the code to only do the comparison for that number of datasets. If you set a limit to 0 or a number larger than the number of available datasets the program will do a full run for all available datasets.

-datasets

This argument allows the user to only perform the for a limited set of datasets for testing the inconsistencies only for those datasets. This option was requested by members of the LDCS team to allow running the tool in batches, which is preferential for users with limited access to the storage locations. This argument expects a comma-separated list of datasets that the user wants the code to compare. The argument datasets and the argument scopes can not be used at the same time.

3.1.2 Running the comparison tool

The flowchart for a visualization of how the tool functions can be found at 10 and the code for the tool can be found at 1 and 2.

I: Retrieve the RSE and scopes

Using the CLI command

```
rucio list-rses
```

```
rucio list-scopes
```

- To check that the input from the user is a valid one.
- If the user chooses to check all the scopes or all the RSE the code will just reference the list of all available scopes/RSE.

These two functions also mean that the code is always aware if any scope or RSE has been added or removed from Rucio, making the tool more future-proof.

II: Retrieve datasets for each scope

Using the command

```
rucio list-dids --filter type=DATASET SCOPE:*
```

allows us to get all the datasets registered to a particular scope. By looping this command for all chosen scopes we get a list of all the datasets. In the case where two scopes share a dataset, the list of datasets is then cleaned for duplicates.

III: Retrieve files in scopes by searching using RSE

Using the command Rucio list-file-replicas –rses RSE DATASET we can get a list of all the file copies listed in the dataset. The flag –reses allow us to limit the search only to files located at a particular location. In the original implementation of this function, all the copies of files in the dataset were requested and the files for a specific RSE were extracted afterward. This was substantially slower than the current implementation because now Rucio will quickly return an empty list if no files at an RSE are registered in a dataset, which means we don't need to check all of the output for files at an RSE in a dataset that has none. However, this change does limit the speed of the program in the case we decide to implement the option to run the tool for multiple RSE at the same time because it will have to request files from Rucio once for each RSE separately.

The output of this command is a string object containing different items of information about the file, so it is converted to a list of different data points regarding the files registered in Rucio. The program also counts the number of files registered to a particular RSE for all datasets and saves this data in a dictionary, information that can be used for classifying the dark data by the analysis program. The program also removes any duplicate output lines.

IV: Clean the data

One disadvantage of using the CLI manual python integration is that the output is designed to be readable in a terminal and is not easily read by python. My solution reads the CLI output line by line, which include lines that do not contain any information related to the files. This means the output from the rucio list-file-replicas command has to be cleaned before it can be used and this is done with a cleaning function. The output goes through several cleaning steps, such as step that removes lines not related to the files, such as the line containing the name of each column in the output and decoration lines, and steps that remove empty list entries and remove parts of strings that do not contain relevant information.

V: Try accessing the files for checksum/count

The LDCS Rucio implementation uses an Adler32 checksum algorithm to verify the integrity of the files and for each file, the Adler32 checksum is registered in hexadecimal. For our code to check the validity of a file, it needs to open sed files and run an Adler32 calculation algorithm. The code is set up in a way where for a given file it fails to open the file it is assumed that this is due to the file not existing in storage and therefore it is flagged as nonexistent. If the code can open the file in storage it runs the checksum algorithm for that file and saves the output to a list as well as denotes the file as present in storage.

If the user chooses to, the program can be set to only check the existence of the files and not the integrity of the file causing the checksum argument when initializing the code. In such a case the program will only attempt to open the file, does not perform a checksum calculation, and records the files it was able to open as present and the rest as missing. During this process, the directories where the files are listed are extracted and added to a dictionary for use when getting a list of all files in storage.

If the checksum operation is performed, the output from the Adler32 algorithm is compared with the checksum registered in Rucio, and any files where there exists a mismatch are flagged as corrupted.

VI: Find files missing from Rucio

From the directories where the files are located, provided by the script in the previous step, the filename for all the files present at these directories are extracted using the ls bash command. If a directory exists that has files registered to in Rucio but is not accessible using the ls bash command, either due to the user not having read permissions for that directory or if the directory does not exist in storage, that directory is put into a list of problem directories Then, by computing the difference between the found files and the files in storage we will find the files that are present in storage but that are not registered in Rucio. These missing files are saved to a separate list.

VII: Output

At this step the code outputs the information gathered from the run so that the user can use this information in the future; either by itself or after it has passed through the classifier tool. The code outputs this information as a result of a complete run for the comparison tool:

The list of files that exist in storage but are not present in the Rucio catalog.

The list of files that exist in the Rucio catalog but are missing from storage.

The list of files that exist both in storage and in the Rucio catalog.

The list of files for which the Adler32 checksum of the file in storage did not match the checksum registered in the Rucio catalog

The list of all the datasets that were examined and the number of files registered to a particular RSE were present in that dataset.

The list of directories that the code was not able to open

3.2 Classification tool

The code for the tool can be found at 3. The classification tool uses the data generated by the comparison tool to classify the dark data and other problem files. This information can then be used by the LDCS team to scan for any systematic problems with the setup. The tool is designed to only analyze output directories from the comparison tool that have performed a full search of the RSE, meaning they have run that tool for all scopes (scopes="All") and with no limit (limit=0). This is to prevent miss classification as limited runs of the comparison tool might not produce enough information in the output files for a certain result.

3.2.1 Retrieve information about the output of comparison tool

The tool has to extract the information about what output directory from the comparison tool we want to perform the check for. The classification tool is set up so that it keeps track of what output folders it has analyzed before and will not rerun the classification tool for them. It is possible to disable this feature by removing the corresponding entry in the "setting.txt" file that keeps track of the comparison to output directories already analyzed.

The tool then needs to extract the individual output files for each output directory. The comparison tool can be run with several different settings, so we can not expect that all the types of output files exist. For example: if the comparison tool is run with the checksum setting set to 'False' the tool will not compare the Adler32 checksum between Rucio and storage, meaning there will not exist an adler32_fail.txt file containing information about the files that failed the checksum. This means the classifier tool has to take this into account when performing all its functions and never assume that a file exists.

3.2.2 Running the tools

The purpose of the runner function is to run other classification functions depending on what files exist for complete runs. As mentioned above, not all possible output files for the comparison tool are always present, so the purpose of this function is to determine what classification functions to run.

3.2.3 Classification of files missing from storage

This tool attempts to classify the files that exist in the Rucio catalog, but do not exist in storage. The first classification it performs is it attempts to find if the files belong to a dataset that might have some problem. This is done by counting the occurrence of each dataset among the files missing from the storage and then retrieving the total number of files registered to the dataset, which is provided by the comparison tool. We then calculate this fraction

$$\frac{\text{Occurrence of a dataset for files missing in storage}}{\text{Occurrence of a dataset for all files registered to a particular RSE}}$$

for each dataset, we find the files missing in storage. We then perform a classification based on these criteria:

I. If this fraction is larger than 0.2, meaning that at least 20% of the files in a particular dataset are missing from storage, that dataset is marked as suspicious. This limit has to be low enough that if there exists some systematic problem with a dataset that it is registered, but high enough that problems not associated with the particular dataset but happen to due to other reasons do not trigger this classification. The limit of 0.2 was chosen somewhat arbitrarily based on what I thought would be a reasonable limit but it could be changed in the future if it is discovered it wrongly classifies datasets.

II. If the fraction is equal to 1, meaning that all the files registered to a particular dataset are not present in the storage, we mark that dataset as missing.

III. If the fraction is less than 0.2 we assume that there is no problem with the dataset that could explain why the file is missing from storage, so we have to investigate those files further.

For files that only fulfill the criteria III further examination is needed. The tool retrieves information about directories that the comparison tool had a problem accessing, either due to lacking permissions or due to that directory missing from storage, and checks if any of the files in category III belong to such a directory. If they do belong to such a directory they are marked as such and any files that passed the dataset test and the directory test are marked as having an unknown reason for their existence.

3.2.4 Classification of files missing from Rucio

This tool attempts to classify the files that are not registered in Rucio. A known issue with the LDCS setup is that simulations sometimes output multiple files for a single simulation, which they are not supposed to do. In such a case only some of the files might be registered to Rucio, and the rest will only exist in storage. A file created by a simulation has this general naming convention:

$$\underbrace{\underbrace{\text{mc\_v9-8GeV-1e-target\_photonuclear}}_{\text{Unique for every simulation run}}\_\overbrace{14569}^{\substack{\text{Unique number for each}\\ \text{file in simulation run}}}\_\underbrace{t1589280908}_{\text{timestamp}}.root}_{\text{Unique for every file}} \tag{2}$$

When multiple output files are created for a single simulation they have the characteristic that they all share the same file name up until the timestamp, which we will refer to as Filename Before Timestamp (FBT), which normally should be unique for every file. This is exploited to classify the files.

The first step is to compare the FBT between the files missing from Rucio, as there might be multiple files not registered in Rucio among the duplicates. We then compare the FBT of the files missing from Rucio with the files that exist both in Rucio and in storage, to find the original file and any duplicate files that somehow were able to get registered to Rucio. For each set of files that share FBT, we calculate their Alder32 checksum. If the checksum matches between them then they are simple copies, otherwise the duplicate is not a copy and it is unknown which file, if any, denotes the actual simulation data. Any files that did not have any duplicates among the files missing in Rucio or among the files both in Rucio and in storage are marked as just missing.

4 Dark data search and analysis at LDCS

4.1 Running our own simulations campaign: trying to understand dark data generation

Dark data appeared at the LDCS during the large simulation campaign that was performed during 2020-2021. As part of the bachelor thesis, we want to run a similar, but smaller scale, simulation campaign to study the conditions for dark data generation. During this simulation campaign, we were using LDCS in the same way as the large-scale campaign before to perfectly replicate the conditions that led to the dark data we now have. Bellow, we describe the simulation workflow used for generating a background signal efficiency estimation.

4.1.1 Small scale simulation campaign

As mentioned in section 2.3 to run a job, a configuration file has to be produced that specifies the simulation parameters. Because we are gathering data for a background estimation we bias those processes in the simulation with a factor of 450. This means that our simulation is 450 times more likely to produce a background photon that then reacts photonuclear in the ECal detector than it would normally be expected. This is later corrected, meaning we do not change the physics of such a simulation, just gather more statistics. We have to specify what processes we want to bias, and in our case, we want to create a bias for the photon particle interacting in a photonuclear interaction inside the ECal. It was requested by the LDMX team that we simulate at least a million events for them to have enough data for their analysis, so we agreed to run a simulation resulting in approximately 10 000 000 events. To determine the number of simulations we would need to perform to get this number of events we submitted several test jobs and gathered statistics on how many events we got per job. The average number of events per job we got was 8071, meaning we had to perform at least $\frac{10000000}{8071} \approx 1239$ jobs to have a satisfactory number of events. For safety, we ran 1300 jobs with one of the jobs failing, resulting in 1299 output files.

To calculate the momentum of an electron traveling through the tagging tracker reconstruction of the tracks in the detector is required, which is a computationally intensive task. A triggering system has to be used to disqualify events not relevant to our search for dark matter. As mentioned in the description of the ECal detector 2.1 we can use the criteria that require the electrons to have less energy than in the beam and we assume a reaction with the target that resulted in undetected particles has happened and keep this event for reconstruction.

4.1.2 Dark data appearance during reconstruction

To preserve roughly the same size of the input and output files, which is done to prevent overloading the LDCS system with many small files, we combined multiple simulation output files into a single reconstruction job. This was done by running a test job, looking at how large the output file was and checking how many events survived the triggering, and based on these numbers adjusting the number of files per reconstruction. In our case, the number we chose was 14 simulation files for each reconstruction job, meaning that from our 1299 simulation files we expect $\frac{1299}{14} \approx 93$ jobs, which is what we saw in the LDMX front end portal. With one output file per reconstruction job, we would expect 93 output files, but in the output directory, there were 111 files present. It was found that file 1-18, meaning file in with the naming scheme 2 where the unique number was 1-18, had a duplicate output file.

4.2 Search and analysis of generated dark data at LDCS storages

Based on the results from running the comparison tool from the DDS toolkit 3.1 LDCS storage's there was dark data generated during the big simulation campaign 2020-2021. This dark data is occupying valuable space on LDCS storages. As part of this thesis, we run a systematic discovery, using the DDS toolkit we developed, of this dark data and classified the files based on our guesses for their origin. We ran the comparison tool 3.1 for all the files registered to storage in Lund and storage in Lund with GRIDFTP access. We ran a more limited search for the storage at SLAC and only looked for dark data in files registered to the **mc20** scope, due to computational limitations imposed on the user by the storage system. We also ran a checksum comparison for all the files that were found at both the storage in Lund and registered in the Rucio catalog but chose not to run these tests for the other storage locations due to the length of these checksum comparisons, as explained in further detail in section 6.1. The classification tool 3.2, which is used to group different dark data files, was run for all the dark data files that were discovered during our search campaign described above.

From the result of running the classification tool for the storage at Lund, storage at Lund with GRIDFTP remote access, storage at SLAC, and storage at SLAC with GRIDFTP remote access, it was determined that the duplicate files were not simple copies of each other due to a difference in the Adler32 checksum. Therefore it was assumed that all or all but the original file had been somehow changed in the process of creating these duplicates, and it was assumed that those duplicate files were corrupted and unreadable. This assumption was proven false when we compared the data from two duplicates for a single file and found that both the files were valid root files and contained very similar information. For the duplicates with file number 1 we compared the number of registered events and found that they had 8058 and 8055 events, a difference that is unluckily to occur due to corruption of a file. We decided to continue our analysis of these files so the BackEnergy plots of the two duplicates overlaid on top of each other can be seen in Figure 4. A similar plot was produced for duplicates of the second file, and the result can be seen in Figure 5. Comparing these two plots, we see that the lines for the two duplicates overlap while simultaneously there is a major difference between the plot for Figure 4 and Figure 5. This seems to indicate that the duplicates are somehow related to each other, for instance, reruns of the same reconstruction job because otherwise, they would be no reason for two duplicate files to be more similar to each other than what we would expect comparing two files at random.

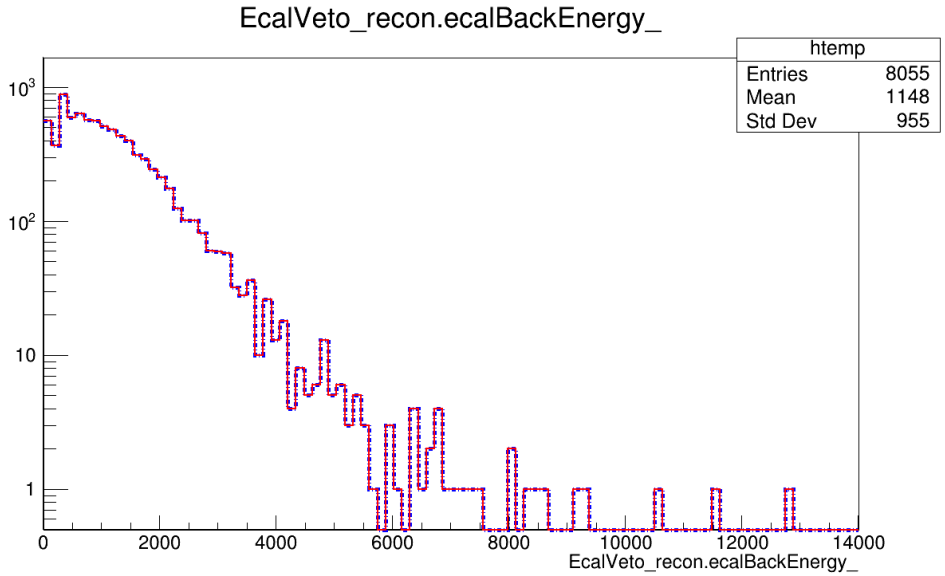


Figure 4: Overlay of two plots for the duplicate of file number 1. The earlier arriving duplicate file is marked in a blue dotted line and the later arriving duplicate marked in a red solid line.

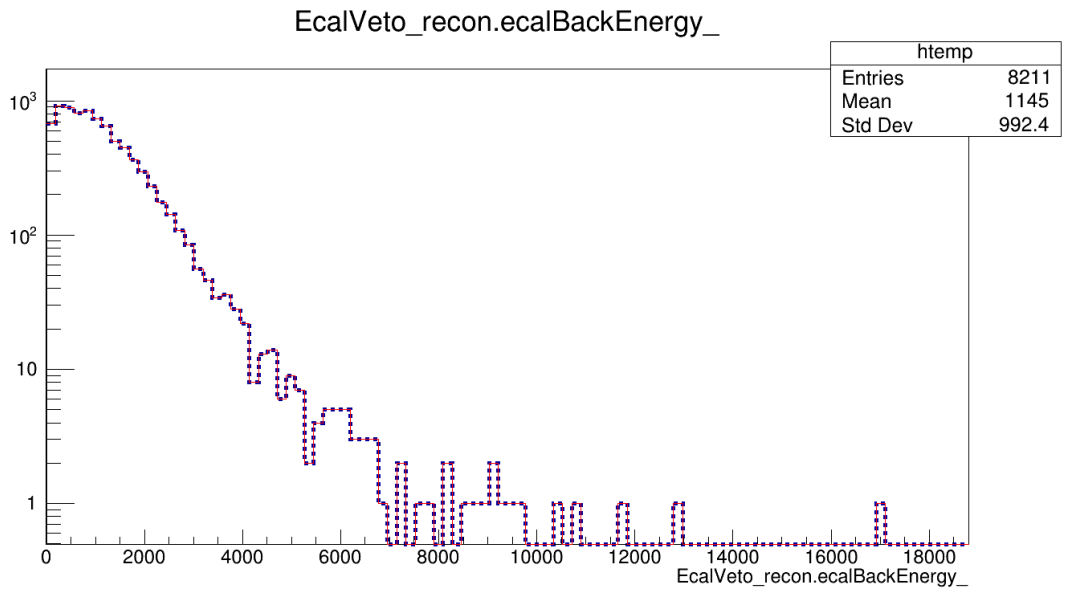


Figure 5: Overlay of two plots for the duplicate of file number 2. The earlier arriving duplicate file is marked in a blue dotted line and the later arriving duplicate marked in a red solid line.

Because the difference in time between two duplicate files increased with the file number (see Figure 6) we wanted to compare two duplicates for a higher file number, to see if the patterns we have seen so far still hold. For this, we selected the duplicates for file number 17 and we got the BackEnergy plot in Figure 7.

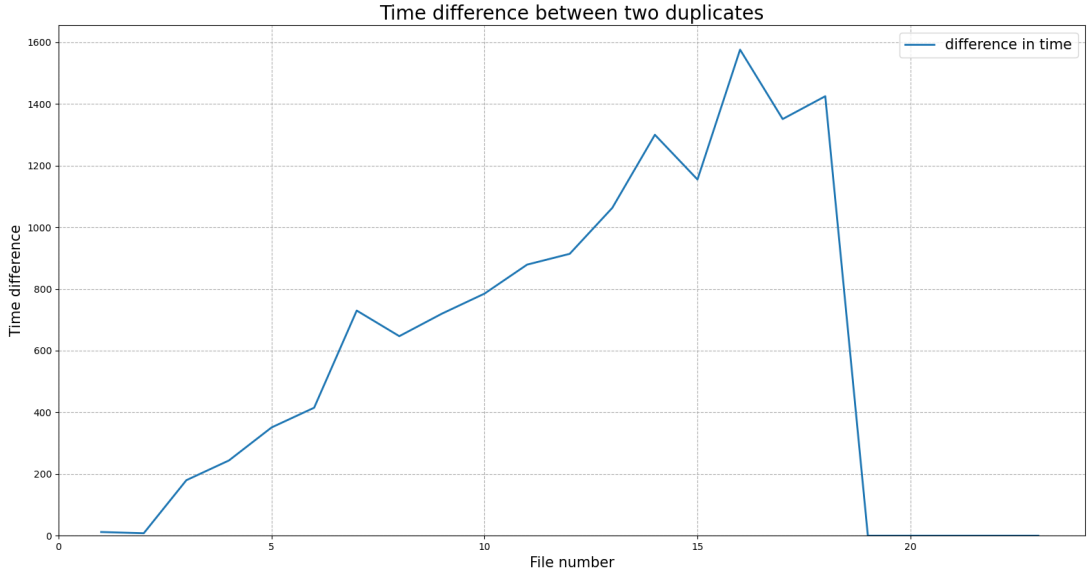


Figure 6: Plot of the time difference between two duplicates from table 6 plotted against the file number.

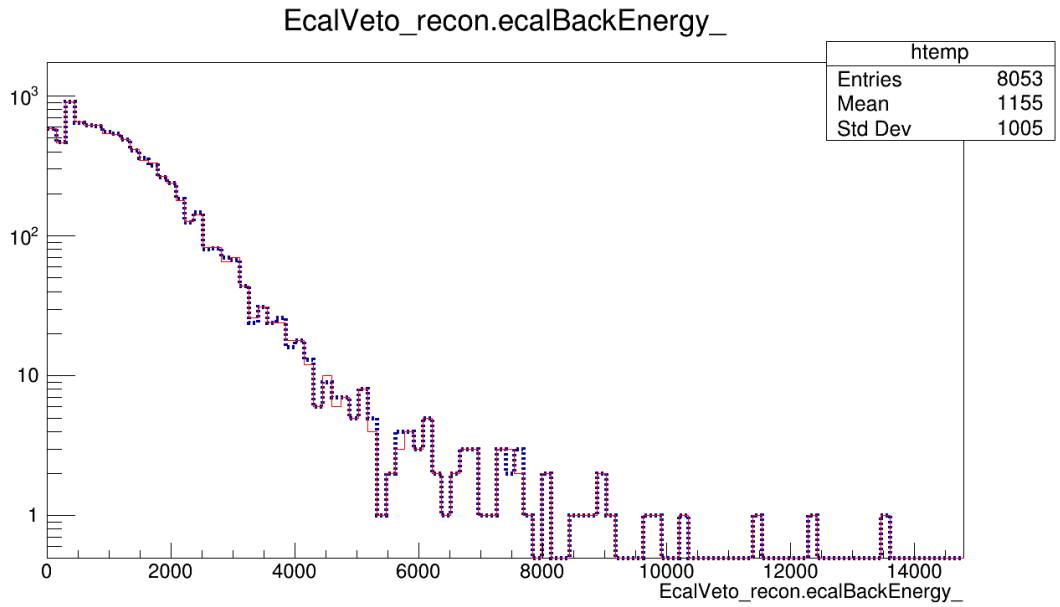


Figure 7: Overlay of two plots for the duplicate of file number 17. The earlier arriving duplicate file is marked in a blue dotted line and the later arriving duplicate marked in a red solid line.

We can see in Figure 7 that the difference between the two lines, the red line for the later duplicate and the blue dotted line for the earlier duplicate, is substantially larger then for Figure 4 and Figure 5, which seems to indicate some correlation between the time difference between duplicates and the amount of difference between the duplicate data. To study this further we plotted the relationship between the difference in time between duplicates against the file number, and the result can be seen in Figure 8.

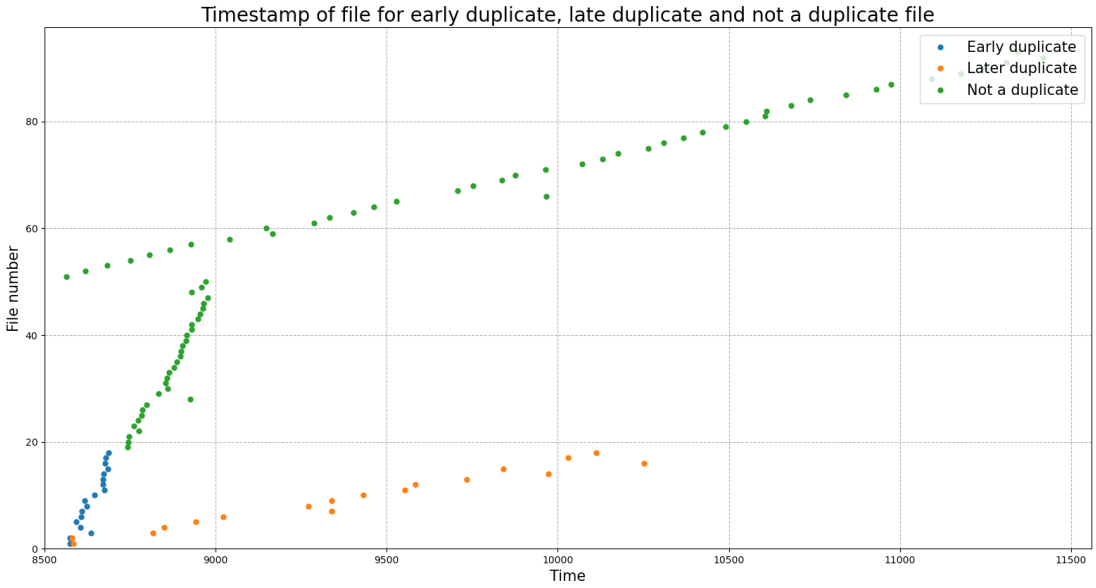


Figure 8: Plot of the time for the early duplicate (blue dots), the time for the later duplicate (orange dots) and the time for the files with nu duplicates (green dots), provided from table 6.

In Figure 8 we see two trends based on the value of the timestamp and the frequency of the timestamp:

Trend I. The duplicate with the earlier timestamp (blue dots) and the files that do not have a duplicate (green dots) up until around the file numbered 50.

Trend II. The duplicates with the later timestamp (orange dots), which end at file numbered 18, and the files that do not have a duplicate (green dots) starting at around file numbered 50.

A theory to explain this, that was discussed by the LDCS team, was that the same reconstruction job was submitted to two different computational centers, in our case Lund and SLAC. The job is completed at both centers and moved to storage at Lund, but due to the proximity of Lund's computational center to the storage in Lund compared to the distance needed to get the data from the computational center at SLAC, San Francisco, we get a later timestamp for the files arriving from SLAC and would also get a slower rate for the timestamp. Based on the result we had received at that point several theories were suggested to explain why we only had duplicates for consecutive files with low numbers, but research into similar simulations we performed at a later date revealed similar consecutive duplicates but starting at larger values for the file number.

The discoveries we made concerning the duplicates are based on surface-level research into the new information and more thorough research is needed. The purpose of this bachelor thesis was to perform a quantitative look into the issue of dark data, including the phenomenon of duplicate files, so this kind of thorough research is outside the scope of this thesis.

4.3 LDCS shortcomings discovered during the dark data search

One purpose of this bachelor's project was to probe the LDCS setup for systematic issues, and we successfully found several instances where current procedures have been insufficient or other technical problems with the LDCS.

One issue that was discovered early on in the process, was that datasets were deleted from storage but kept registered in Rucio. While this was a known problem due to the lack of a system for automatic deletion of the Rucio entries for files deleted from storage, the extent of this problem was not known. As an example for the storage location at SLAC with GRIDFTP remote access 15 we found that multiple directories were missing from storage that was registered in Rucio. After discussion with the LDCS team, this was verified as something that was manually deleted from storage, but they forgot to register to Rucio.

A different problem discovered by the tool that was not originally known by the LDCS team was several directories storing data at storage location LUND had the permissions set to only accessible by the original creator 16. This was verified by the LDCS team and raised the question of how file permissions were set at the creation of the files by ARC and modifications to the current setup were proposed.

Another problem discovered was a number of files were registered to a directory that did not exist exist at the storage location, Lund:

/nfs/ldmx/ldmxstore/ldmx/mc-data/mc20/v12/4.0GeV/v2.2.1-batch22/

This discovery was brought forward to the LDCS team and it was noted that the address resembles an address that would be found at SLAC storage. After further research, it was established that this was not an error with the tool but somehow files were registered to another storage location than where they were located, which seems to indicate some error in how the file location is matched to the storage center in the Rucio catalog.

Lists of other found problems can be found in table 2, 1

5 Results

In this section, we present the results of the dark data search and analysis that was performed by running using the DDS toolkit for the storage in Lund and at SLAC.

For the storage location at Lund, the DDS tools discovered two files that had been corrupted 14. This is equivalent to $\frac{2}{419090} = 4.77224 \cdot 10^{-6}$ %, which is a minuscule percentage. While a complete run, meaning a run that tool for all scopes (scopes="All") and with no limit (limit=0), with Adler32 checksum comparison was not complete for the storage at Lund with GRIDFTP access due to limitations set by LUNARC, it was able run the checksum comparison for more than half of the number of files at that storage and did not discover any corrupted files. Based on this result combined with the issues other LDCS members had running the checksum tool due to using up too much of their time-limited access to the storage site, it was decided not to perform this comparison for the storage at SLAC and the storage at SLAC with GRIDFTP access. Even though the program did not discover a large number of corrupt files, before running my DDS toolkit no information regarding such files was available to the LDCS team. Given what we know now, there is no need to worry about corrupted files in storage.

As can be seen in table 1 the code discovered that the files belonging to the runs **mc_v9-8GeV-1e-target_photonuclear** and **mc_v12-4GeV-2e-ecal_photonuclear** had a larger than 100% of the files missing in Rucio be duplicates, meaning that for each file belonging to these runs in Rucio there exist more than one duplicate in storage and Rucio. This was originally believed to be an error related to the classification tool, but after further research, it was discovered that there existed a large number of duplicates registered to Rucio, sometimes in the hundreds for files with the same FBT 2, which was not something that was previously believed to be possible for the duplicate files. This discovery was brought forward to the LDCS team and it was noted that these may be duplicates that were created due to a problem that arose when support was added for multiple input files per job. This issue was caused by a change to the code governing the ACT made on Jul 5, 2021, and was resolved on Nov 16, 2021. Checking these simulation runs we found that the files were moved to Lund storage with GRIDFTP access on these dates:

- **mc_v9-8GeV-1e-target_photonuclear** moved on Sunday 19 Sep 2021
- **mc_v12-4GeV-2e-ecal_photonuclear** moved on Wednesday 17 Nov 2021.

This seems to indicate that while **mc_v9-8GeV-1e-target_photonuclear** could be explained by the problem related to multiple input files per job, the run **mc_v12-4GeV-2e-ecal_photonuclear** can not. This has to be researched further but is outside the scope of this paper.

From our results in table 4, we see that only the storage in Lund without any GRIDFTP access has a significant number of files missing from storage. While this issue was known to exist for the storage in Lund, the lack of this issue at other storage locations was not. The same situation concerning the classification of files missing from Rucio, where only for the Lund storage location was there a significant number of files not registered to Rucio that were not a duplicate. Another problem that was known before the beginning of the project was that a large number of files would be missing from Rucio for the storage at Lund with GRIDFTP access, but the extent was substantially underestimated. This discovery however means that the analysis of the storage done by the DDS toolkit can result in, if the dark data files are deleted from storage, over 40 TB of saved storage space.

A conclusion that could be reached from the research conducted in this thesis regarding the dark data, is that the vast majority of it was created manually due to deletion of the data from storage but no deletion of the Rucio catalog entries for those files, or vise versa. The DDS tools now provide a reliable and relatively fast way to discover those files and they can not easily be deleted by looking at the output txt files from the comparison tool. Another conclusion I make is that the issue of duplicate files was substantially underestimated, as it was originally believed to be an extremely rare occurrence, in the region of a few hundred. Now we know that for the three data locations (Lund, Lund with GridFTP and SLAC with GridFTP for the scope mc20) from table 5 we have:

$$\underbrace{589}_{\text{LUND}} + \underbrace{21673}_{\text{LUND\_GRIDFTP}} + \underbrace{1699}_{\text{SLAC\_GRIDFTP}} = 23961 \text{ duplicate files}$$

6 Outlook

6.1 Missing features and bugs.

The DDS tools we created are a complete solution to the problem of discovering dark data and it fulfills the task of comparing the Rucio catalog and the files in storage for inconsistencies. Nevertheless, there still exist several features that have been requested by the team or features that I think would be a good addition to the current tools that we did not implement during the development of this version. There also exist several bugs/problems with the tools that do not affect the functionality but may affect the performance and time spent running the code.

Too many files

When running the code for all the files registered in Rucio for the RSE Lund, meaning running this command in the rucio-client environment

```
python DDS_comparison_tool.py rse="LUND" scopes="All" output="True" limit=0 checksum="True"
```

the code discovered 1180566 files, which is substantially larger than the number of files that we expect to find registered in Rucio or the Lund RSE.

From the result section, we can see that the code found that there are 1346 files registered in Rucio that are missing from storage and 1054 files in storage that are missing in Rucio. We also find that Lund storage holds a combined 420144 files that are associated with LDCS. This means that the Rucio catalog should only have this many files registered for the Lund RSE:

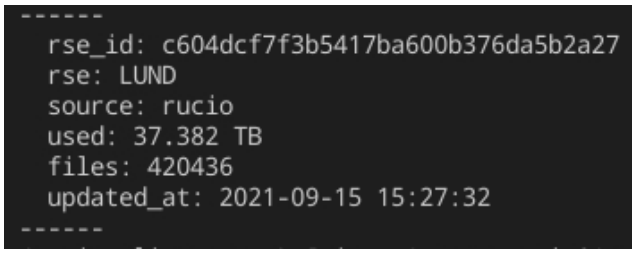
$$\text{Number of files Rucio} = \text{Number of files storage} - \text{files storage but not Rucio} + \text{files Rucio but not storage}$$

$$420436 = 420144 - 1054 + 1346$$

This is consistent with what we get if we use the Rucio CLI command

```
rucio list-rse-usage LUND
```

which returns that the RSE Lund has 420436 files registered to it in Rucio.



This means that the code finds $1180566/420436 \approx 2.8$ times as many files in Rucio for the Lund RSE than there actually should be. Most likely, this is due to files being registered to multiple datasets, which would mean we would search for a file twice. This problem was noticed early on in the process, but I have been unable to remove it due to the need to keep statistics on how many files exist in every dataset, which means keeping only one copy of a file might not count them correctly when classifying the dark data using the number of files registered to each dataset. A similar situation was seen for LUND_GRIDFTP and SLAC_GRIDFTP, but due to running the DDS tools a while ago the number of files registered to those RSEs will not match what we got from the DDS due to new files being added to both Rucio and storage since then.

This is conceptually easy to solve, as it can be solved using some sort of dictionary where each file name can correspond to multiple sets of information from different datasets, but implementing and testing such a solution would be time-consuming and we did not have the time to implement such a function into the code.

Speed of Adler32 checksum calculator

The tool spends the majority of its time calculating the Adler32 checksum for the files in storage, an operation that for a full run for all the datasets with the RSE Lund takes 195.9 hours, which is over 8 days. Any decrease in the time it takes to calculate the Adler32 checksum for a file would have a large effect on the total computational time. To implement a faster Adler32 algorithm, I would have needed to test different available algorithms for speed and to make sure that they function properly, which would take development time that I did not have.

No remote GridFTP access

Currently, the tools I have created do not support the use of the GridFTP file transfer protocol for accessing the files at remote locations. This means I can only run the tool locally even though a remote option exists for the storage locations:

LUND storage with GRIDFTP access (RSE: LUND_GRIDFTP)

SLAC storage with GRIDFTP access (RSE: SLAC_GRIDFTP)

This would take significant effort and time to implement, and it was deemed not a priority while designing this tool.

6.2 Future work

The discoveries we made concerning the duplicates are based on surface-level research into the new information and more thorough research is needed. The purpose of this bachelor thesis was to perform a quantitative look into the issue of dark data, including the phenomenon of duplicate files, so this kind of thorough research is outside the scope of this thesis. This would include searching for more patterns regarding the duplicate files, such as the conditions for the creation of the files like the computational site, time of creation, number of failed simulation attempts, and how busy the sites were at the time of the creation for the duplicate files. A new study has been discussed by the LDCS teams and a project proposal is being drafted.

There have also been requests for features to add to the DDS tools, for example, an automated system for splitting the comparison into smaller batches, and the bugs mentioned in the previous section have to be addressed. There have also been discussions about integrating this solution as part of the Rucio framework, which would allow for the detection of dark data without the need of a dump of all the file locations in storage and allow the implementation of checksum comparisons using the Adler32 algorithm or other similar algorithms. A new project has been discussed as part of the LDCS for the implementation of these features.

7 Acknowledgements

I want to thank all the members of the LDMX and the LDCS team for helping me along the way and allowing me to be part of this incredible project. In particular, I want to thank:

Ruth Pöttgen

Senior lecturer at Particle Physics, Lund University

Balazs Konya

Researcher at Particle Physics, Lund University

Lene Kristian Bryngemark

Ph.D. department of Physics, Stanford University

Florido Paganelli

Research engineer at Particle Physics, Lund University

David Cameron

Researcher - High Energy Physics, University of Oslo

Paul Weakliem

CNSI Research Computing Support, University of California Santa Barbara

Erik Wallin

Master student at Particle Physics, Lund University

8 Figures and tables

Metadata record for simulations

attribute name	type/format	example	source	description
SampleId	string	v9-8GeV-1e-ecalpn	config	The name of the sample the file belongs to
IsSimulation	yes/no	yes	Known by origin	Specifies whether the data file contains simulation data or detector data.
LdmxImage	string	ldmx-v1.7.0-gLDMX.10.2.3_v0.3-r6.18.04-el7-d7.sif	Determined from the RTE specified in the config	The name of the image file containing the LDMX software stack
ComputingElement	string	l-pilot.lunarc.lu.se	Comes from ACT	The name or ID of the computing cluster that generated the data
JobSubmissionTime	timestamp		Comes from ACT	The timestamp when the job was created in the production system
FileCreationTime	timestamp		cluster	The date/time when the simulation output file was created
Walltime	number	2358789	cluster	Walltime minutes used to generate the data
ARCCEJobID	id		cluster	The local ID of the simulation task within an ARC CE
BeamEnergy	number	8.0	Config and mac	Beam energy specified in GeV.
ElectronNumber	integer	1	Config and mac	Number of electrons in a bunch.
PhysicsProcess	string	ecal-photonuclear	Config and mac	The simulated physics process.
DetectorVersion	string	v9	Config file	Version of the detector design used in the simulation.
MagneticFieldmap	string	BmapCorrected3D_13k_unfold	Config file	The name of the file containing the magnetic field info

Figure 9: An example of the metadata generated from a simulation.

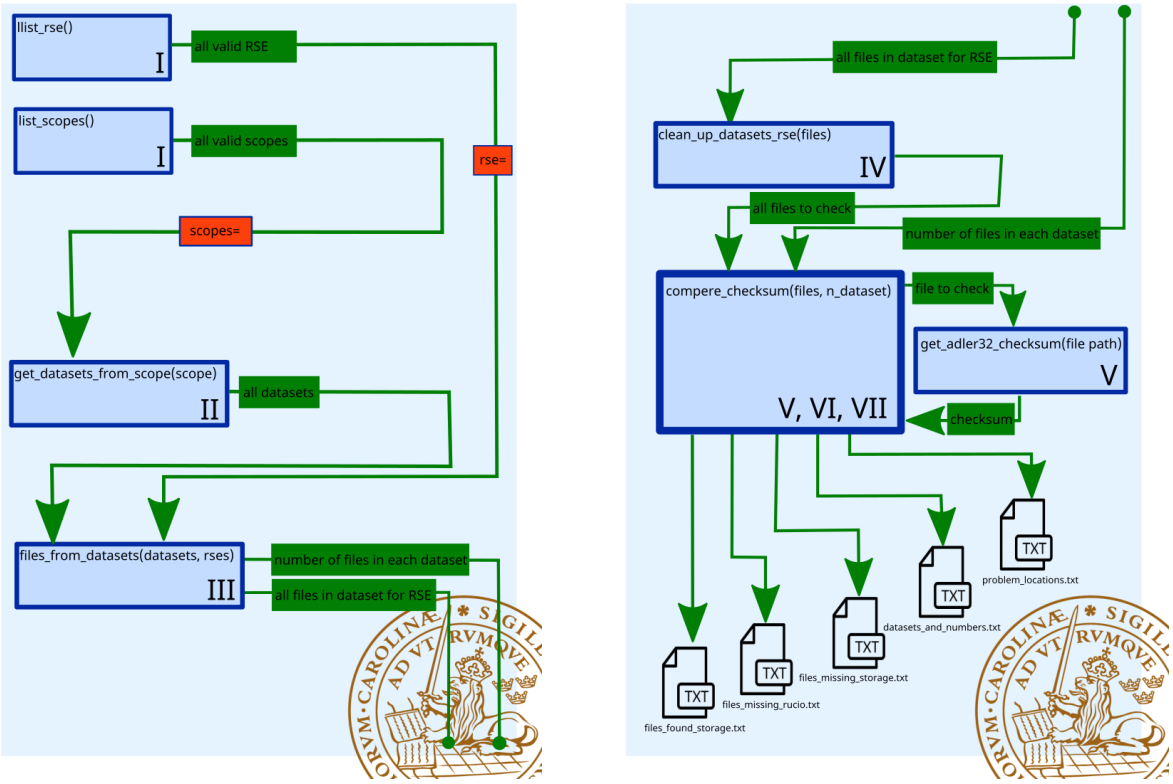


Figure 10: Flowchart for the comparison tool where blue boxes denote different functions, green lines and boxes denote output and input information for the different functions and the red boxes denote changes to functions input based on arguments provided to the classification tool.

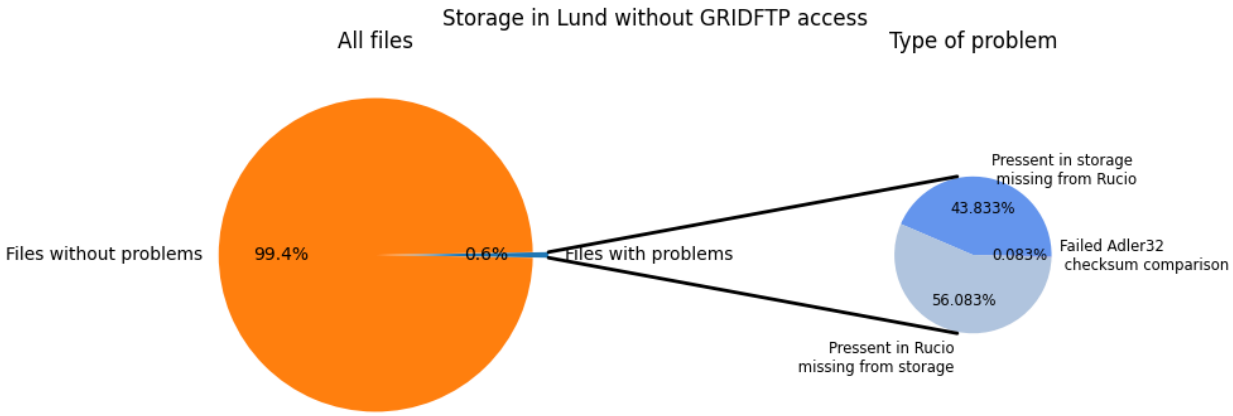


Figure 11: The left pie plot is of all the files registered and located at storage in Lund. The blue section represent all files that were flagged as dark data and the distribution of the type of dark data can be seen in the right pie plot.

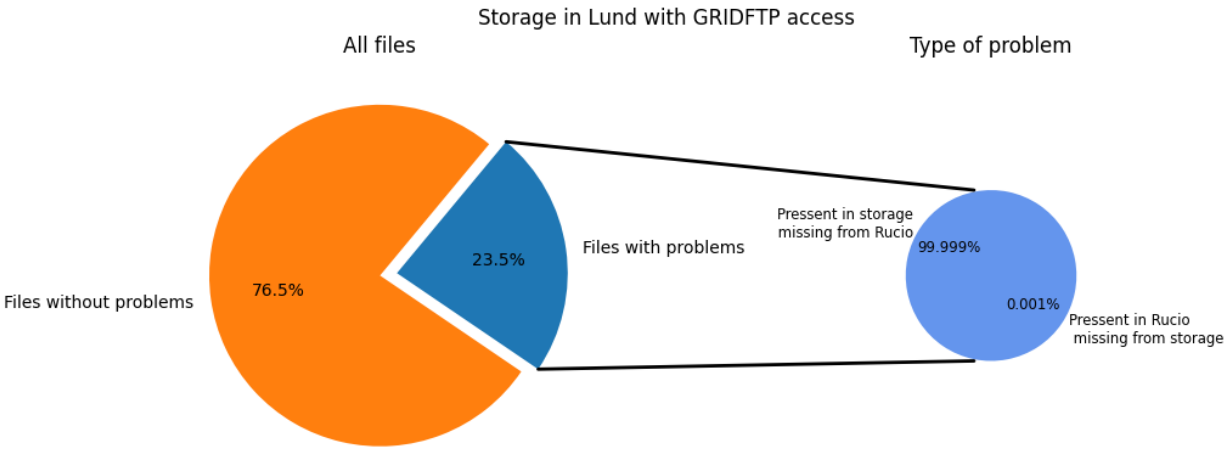


Figure 12: The left pie plot is of all the files registered and located at storage in Lund with remote GRIDFTP access. The blue section represent all files that were flagged as dark data and the distribution of the type of dark data can be seen in the right pie plot.

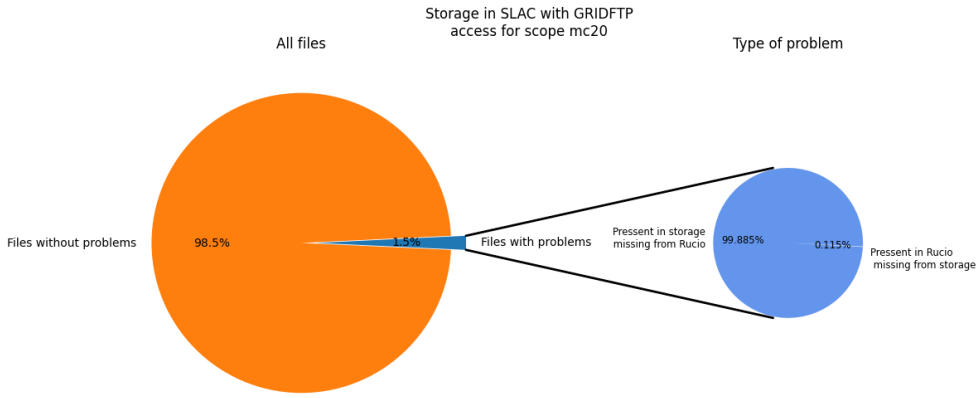


Figure 13: The left pie plot is of all the files registered and located at storage at SLAC with remote GRIDFTP access. The blue section represent all files that were flagged as dark data and the distribution of the type of dark data can be seen in the right pie plot.

```
mc_v12-4GeV-1e-ecal_photonuclear_run1202033_t1601104928.root,4c3f54ac,594eaefd
mc_v12-4GeV-1e-ecal_photonuclear-test_t1594763669.root,7077a49f,ff8cef29
```

Figure 14: An example output regarding files that failed the Adler32 checksum comparison.

Table 1: Output from the classifier tool that summaries the problems found at the RSE LUND_GRIDFTP.

LUND_GRIDFTP
The run **mc_v9-8GeV-1e-target_gammamumu** in Rucio might have some problem, because 34.16 % of the files missing in Rucio are duplicates.
The run **mc_v9-8GeV-1e-target_photonuclear** in Rucio might have some problem, because 175.93 % of the files missing in Rucio are duplicates.
The run **mc_v12-4GeV-2e-ecal_photonuclear** in Rucio might have some problem, because 112.38 % of the files missing in Rucio are duplicates.

Table 2: Output from the classifier tool that summaries the problems found at the RSE LUND.

LUND
The batch **mc-test20:v9-8GeV-1e-inclusive** is present in Rucio but is missing from storage.
The batch **mc20:v9-8GeV-1e-inclusive** is present in Rucio but is missing from storage.
The batch **test:v9-8GeV-1e-inclusive** in Rucio might have some problem, because 75.0 % of it is missing in storage.
The run **mc_v12-4GeV-1e-ecal_photonuclear-test** in Rucio might have some problem, because 99.5 % of the files missing in Rucio are duplicates.
The run **mc_v12-4GeV-1e-ecal** in Rucio probably has some problem, because 100.0 % of the files missing in Rucio are duplicates.
The file **mc_v12-4GeV-1e-ecal_photonuclear_run1202033_t1601104928.root** has been corrupted. The value for the Adler32 checksum in Rucio is 4c3f54ac but the checksum in storage is 594eaefd.
The file **mc_v12-4GeV-1e-ecal_photonuclear-test_t1594763669.root** has been corrupted. The value for the Adler32 checksum in Rucio is 7077a49f but the checksum in storage is ff8cef29.

Table 3: Output from the comparison tool

	LUND	LUND_GRIDFTP	SLAC_GRIDFTP scope mc20
Files found in storage	420142	862599	116701
Files found in storage and Rucio	419090	660145	114957
Failed checksum comparison	2	No check	No check
Missing from Rucio	1052	202454	1744
Missing from storage	1346	2	1

Table 4: Missing from storage classification

	LUND	LUND_GRIDFTP	SLAC_GRIDFTP scope mc20
Missing from storage	1346	2	1
Datasets missing	2	0	0
Number of files in missing datasets	1339	0	0
Datasets with potential problems	1	0	0
Number of files in datasets with potential problems	6	0	0
Files belonging to problem locations	1	0	0
Files not classified	0	2	1

Table 5: Missing from Rucio classification

	LUND	LUND_GRIDFTP	SLAC_GRIDFTP scope mc20
Missing from Rucio	1052	202454	1744
Not a duplicate	439	661	5
Duplicate	589	201673	1699
Problem simulation runs	2	3	0

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch28/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch42/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch44/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch38/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch25/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch26/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch30/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch24/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch29/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch27/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch43/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch40/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch41/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch39/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch46/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch45/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch31/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch36/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch37/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch32/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch34/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch35/

gpfs/slac/staas/fs1/g/ldmx/data/mc20/v12/4.0GeV/v2.3.0-batch33/

Figure 15: Directories at SLAC discovered by the code that were not accessible to the program due to improper permissions set for the files.

/projects/hep/fs9/shared/ldmx/ldcs/output/ldmx/mc-data/validation/v12/4.0GeV/v2.3.0-testReco/

/projects/hep/fs9/shared/ldmx/ldcs/output/ldmx/mc-data/validation/v12/4.0GeV/v2.3.0-1e-test/

/projects/hep/fs9/shared/ldmx/ldcs/output/ldmx/mc-data/validation/v12/4.0GeV/v2.3.0-testRecoLund/

/projects/hep/fs9/shared/ldmx/ldcs/output/ldmx/mc-data/mc20/v12/4.0GeV/v2.3.0-3e/

/projects/hep/fs9/shared/ldmx/ldcs/output/ldmx/mc-data/mc20/v12/4.0GeV/v2.3.0-4e/

/projects/hep/fs9/shared/ldmx/ldcs/output/ldmx/mc-data/mc20/v12/4.0GeV/v2.3.0-2e/

/projects/hep/fs9/shared/ldmx/ldcs/output/ldmx/mc-data/validation/v12/4.0GeV/v2.3.0-testInput/

/projects/hep/fs9/shared/ldmx/ldcs/output/ldmx/mc-data/validation/v12/4.0GeV/EaT-test/

/projects/hep/fs9/shared/ldmx/ldcs/output/ldmx/mc-data/validation/v12/4.0GeV/v2.3.0-batch1/

Figure 16: Directories at Lund discovered by the code that were not accessible to the program due to improper permissions set for the files.

```
mc_v12-4GeV-1e-ecal_photonuclear_run223386_t1608602012.root,/projects/hep/fs9/shared/ldmx/ldcs/output/ld
data/mc20/v12/4.0GeV/v2.3.0-batch23/
mc_v12-4GeV-1e-ecal_photonuclear_run224531_t1608613083.root,/projects/hep/fs9/shared/ldmx/ldcs/output/ld
data/mc20/v12/4.0GeV/v2.3.0-batch23/
mc_v9-8GeV-1e-target_photonuclear_10541_t1589186375.root,/projects/hep/fs7/scratch/pflorido/ldmx-
pilot/pilotoutput/ldmx/mc-data/v9/8.0GeV/
mc_v9-8GeV-1e-target_photonuclear_1081_t1589175256.root,/projects/hep/fs7/scratch/pflorido/ldmx-
pilot/pilotoutput/ldmx/mc-data/v9/8.0GeV/
mc_v9-8GeV-1e-target_photonuclear_11091_t1589138287.root,/projects/hep/fs7/scratch/pflorido/ldmx-
pilot/pilotoutput/ldmx/mc-data/v9/8.0GeV/
```

Figure 17: Example of output file from the comparison tool listing files present in storage but missing from Rucio.

Table 6: Output files from the reconstruction job v1.7.1_ecal_photonuclear_reco_bdt-batch1 where "Time for early duplicate" refers to the file with the earliest timestamp and "Time for late duplicate" refers to the file with the later timestamp (see 2). The time difference between the two duplicates is measured in seconds.

File number	Time difference between earliest and later duplicate	Time for early duplicate	Time for late duplicate
1	12	8585	8573
2	8	8581	8573
3	180	8816	8636
4	244	8849	8605
5	351	8943	8592
6	415	9022	8607
7	730	9339	8609
8	647	9271	8624
9	720	9338	8618
10	785	9431	8646
11	879	9553	8674
12	914	9584	8670
13	1063	9734	8671
14	1300	9973	8673
15	1155	9841	8686
16	1576	10252	8676
17	1351	10030	8679
18	1425	10113	8688
File number	Time difference for files without duplicates	Time for normal file	Time for normal file
19	0	8743	8743
20	0	8745	8745
21	0	8746	8746
22	0	8775	8775
23	0	8761	8761
...	...	...	...

9 References

[1] AAkesson, Torsten, Asher Berlin, Nikita Blinov, Owen Colegrove, Giulia Collura, Valentina Dutta, B. Echenard, et al. "The Light Dark Matter EXperiment (LDMX)." Proceedings of The 39th International Conference on High Energy Physics — PoS(ICHEP2018), 2019.

[2] Rueter, Thomas D., Thomas G. Rizzo, and JoAnne L. Hewett. 'Dark Matter Freeze Out with Tsallis Statistics in the Early Universe'. arXiv, 2019. https://doi.org/10.48550/ARXIV.1911.11254

[3] Raubenheimer, Tor, Anthony Beukers, Alan Fry, Carsten Hast, Thomas Markiewicz, Yuri Nosochkov, Nan Phinney, Philip Schuster, and Natalia Toro. 'DASEL: Dark Sector Experiments at LCLS-II'. arXiv, 2018. https://doi.org/10.48550/ARXIV.1801.07867.

[4] Bryngemark, Lene Kristian, David Cameron, Valentina Dutta, Thomas Eichlersmith, Balazs Konya, Omar Moreno, Geoffrey Mullier, et al. 'Building a Distributed Computing System for LDMX'. EPJ Web Conf. 251 (2021). https://doi.org/10.1051/epjconf/202125102038.

[5] Barisits, Martin, Thomas Beermann, Frank Berghaus, Brian Bockelman, Joaquin Bogado, David Cameron, Dimitrios Christidis, et al. "Rucio: Scientific Data Management." Computing and Software for Big Science 3, no. 1 (August 9, 2019): 11. https://doi.org/10.1007/s41781-019-0026-3.

[6] Aad, G. and others. "The ATLAS Experiment at the CERN Large Hadron Collider." JINST 3 (2008): S08003. https://doi.org/10.1088/1748-0221/3/08/S08003.

[7] "Rucio CLI: Examples — Rucio 1.2 Documentation." Accessed March 3, 2022. https://rucio.readthedocs.io/en/latest/cli_examples.html.

[8] "The Client API Reference — Rucio 1.2 Documentation." Accessed March 2, 2022. https://rucio.readthedocs.io/en/latest/api.html.

[9] "Rucio Client API | Rucio Documentation." Accessed March 2, 2022. https://rucio.github.io/documentation/rucio_client_api/.

[10] "Rucio." Accessed March 2, 2022. https://www.overleaf.com/project/62160af9ff589e74c7a8e7e5.

[11] "Daemon Rucio-Auditor — Rucio 1.2 Documentation." Accessed May 2, 2022. https://rucio.readthedocs.io/en/old-doc/man/rucio-auditor.html.

[12] Merkel, Dirk. "Docker: Lightweight Linux Containers for Consistent Development and Deployment." Linux Journal 2014, no. 239 (2014): 2.

[13] Profumo, Stefano. An Introduction to Particle Dark Matter. WORLD SCIENTIFIC (EUROPE), 2017. https://doi.org/10.1142/q0001.

[14] Agostinelli, S., J. Allison, K. Amako, J. Apostolakis, H. Araujo, P. Arce, M. Asai, et al. 'Geant4—a Simulation Toolkit'. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 506, no. 3 (2003): 250–303. https://doi.org/10.1016/S0168-9002(03)01368-8.

[15] Rene Brun and Fons Rademakers, ROOT - An Object Oriented Data Analysis Framework, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. Meth. in Phys. Res. A 389 (1997) 81-86.

[16] AAkesson, Torsten, Asher Berlin, Nikita Blinov, Owen Colegrove, Giulia Collura, Valentina Dutta, B. Echenard, et al. "The Light Dark Matter EXperiment (LDMX)." Proceedings of The 39th International Conference on High Energy Physics — PoS(ICHEP2018), 2019,Figure 17 page 28.

10 Code

```
from test_func import *
from test2 import *
import sys
import test2
import os

#add search by dataset's

#add a timestamp to dump all the output files to for easier tracking and no overwrite
if __name__ == '__main__':
    valid_rses=list_rse()
    valid_scopes=list_scopes()

    #default values
    global tqmdis
    global comments
    global limit
    global checksum

    tqmdis=False
    comments=False
    limit=0
    checksum=True
    test2.datasets2=None
    for argument in sys.argv:
        #argument 1:rse
        if "rse=" in argument:
            argument=argument.replace("rse=","")
            if "All" in argument:
                test2.rses=valid_rses
            elif len(argument)==0:
                raise ValueError("rse= can not be empty")
            elif argument in valid_rses:
                test2.rses=argument
            else:
                raise ValueError("rse={} is not a valid rse. Choose from {}.".format(argument,
                    valid_rses))

        if "scopes=" in argument:
            argument=argument.replace("scopes=","")
            #argument 2 scopes
            if "All" in argument:
                test2.scopes=valid_scopes
            else:
                argument=(argument).split(",")
                for n in argument:
                    if n not in valid_scopes:
                        raise ValueError("At least one of provided scopes is not valid. Choose
                            from {}.".format(valid_scopes))
                else:
                    test2.scopes=argument

        if "output=" in argument:
            argument=argument.replace("scopes=","")
            if "True" in argument:
                test2.tqmdis=False
                test2.comments=True
            elif "False" in argument:
                test2.tqmdis=True
                test2.comments=False
            else:
                raise ValueError("{} is not a valid setting for the output. Choose between False
                    and True.".format(argument))
        if "checksum=" in argument:
            argument=argument.replace("checksum=","")
            if "True" in argument:
                test2.checksum=True
```

```
        elif "False" in argument:
            test2.checksum=False
        else:
            raise ValueError("{} is not a valid setting for adler32 checksum. Choose between
                False and True.".format(argument))
    if "limit=" in argument:
        argument=argument.replace("limit=","")
        if isinstance(argument,str)==True:
            test2.limit=int(argument)
        elif isinstance(argument,int)==True:
            test2.limit=argument
        else:
            raise ValueError("Limit has to be an integer".format())
    if "datasets=" in argument:
        argument=argument.replace("datasets=","")
        if isinstance(argument,str)==True:
            test2.datasets2=argument.split(",")
        else:
            raise ValueError("Limit has to be an integer".format())
try:
    test2.rses
except:
    raise ValueError("No rses provided")
try:
    test2.limit
except:
    test2.limit=0

if test2.datasets2==None:
    pass
else:
    try:
        test2.scopes
        test2.datasets2
    except:
        try:
            test2.scopes
        except:
            try:
                test2.datasets2
            except:
                raise ValueError("You have to provide at least a scope or a dataset.")

    else:
        raise ValueError("Can not search by scope and by dataset similtaniosly , please use
            only one.")

if not test2.datasets2==None:
    number_of_files_in_dataset={}
    rses=test2.rses.split(",")
    All_datasets=test2.datasets2
    datasets_rse, number_of_files_in_dataset=files_from_datasets(All_datasets,rses)
    datasets_rse=clean_up_datasets_rse(datasets_rse)
    compere_checksum(datasets_rse,number_of_files_in_dataset)
else:
    if test2.scopes==valid_scopes and test2.limit==0:
        test2.All=True
    number_of_files_in_dataset={}
    rses=test2.rses.split(",")
    All_datasets=get_all_datasets(test2.scopes)
    datasets_rse, number_of_files_in_dataset=files_from_datasets(All_datasets,rses)
    datasets_rse=clean_up_datasets_rse(datasets_rse)
    compere_checksum(datasets_rse,number_of_files_in_dataset)
```

Listing 1: The code initialising the comparison tool, written in Python.

```
import os
from zlib import adler32
from datetime import datetime
from tqdm import *
from DDS_comparison_tool import *


#initialise default values
tqmdis=False
comments=False
limit=0
checksum=True
All=False


#gets all the valid scopes registered in Rucio
def list_scopes():
    scopes=list(os.popen("rucio list-scopes"))
    for n in range(len(scopes)):
        scopes[n]=scopes[n].replace("\n","")
    return(scopes)


#For a scope get all datasets registered in rucio for that scope 1.
def get_all_datasets(scopes):
    All_datasets=[]
    #Iterate over all scopes, run function get_datasets_from_scope from scope
    for scope in scopes:
        datasets_scope=get_datasets_from_scope(scope)
        All_datasets=list(set(All_datasets+datasets_scope))
    return(All_datasets)


#For a scope get all datasets registered in rucio for that scope 2.
def get_datasets_from_scope(scope):
    #Rucio CLI command returns the datasets as a str
    datasets_scope=list(os.popen("rucio list-dids --filter type=DATASET {}:*".format(scope)))
    #Removes rows not containing data
    [ x for x in datasets_scope if "|" in x ]
    #Clean the data a bit
    for n in range(len(datasets_scope)):
        datasets_scope[n]=datasets_scope[n].replace("| DATASET       |\n'","")
        datasets_scope[n]=datasets_scope[n].replace("|","")
        datasets_scope[n]=datasets_scope[n].replace(" ","")
        datasets_scope[n]=datasets_scope[n].replace("\n","")
        datasets_scope[n]=datasets_scope[n].replace("DATASET","")
    return(datasets_scope)

#List all registered rse in Rucio
def list_rse():
    rses=list(os.popen("rucio list-rses"))
    for n in range(len(rses)):
        rses[n]=rses[n].replace("\n","")
    return(rses)


#For every dataset provided it gets all file replicas that are located at a particular rse
def files_from_datasets(datasets, rses):

    #remove duplicate datasets
    datasets=list(set(datasets))
    datasets=[file for file in datasets if not "----------" in file]

    #print if output is turned on
    if comments==True:
        print("\nGet a list of all the files in a dataset.")

    #library storing all the files found at a rse where keys are rse and items are a list of files
    datasets_rse={}
```

```
for rse in rses:
    #list of files found for the rse
    list_of_files=[]

    #lsit to keep track of how many files are located at a rse for a particular dataset. This
        info can be used in the future to classify problematic files
    number_of_files_in_dataset={}

    #"if limit==0" is to check if you want to run the version for all the datasets or only for
        some of them.
    if limit==0:
        for n in tqdm(range(len(datasets)), disable=tqmdis):
            dataset=datasets[n]
            #Clean the datasets by removing the first and last line that contains junk
            if "SCOPE:NAME[DIDTYPE]" in dataset or "--------------------------------------------------------------------------------" in dataset:
                pass
            else:
                #Get all the files in a dataset at a rse and save it as a list by splitting at
                    each line
                L=(os.popen("rucio list-file-replicas --rses {} {}".format(rse,dataset)).read
                    ()).split('\n')

                #Clean the data a bit by removing any line not containing the name of the rse,
                    which is present in every file location.
                L=[file for file in L if rse in file]
                L=[file+str(dataset) for file in L]

                #add the number of files with a particular rse to the dictionary
                number_of_files_in_dataset[dataset]=len(L)

                #files to file list
                list_of_files.extend(L)
    else:
        #Same as above
        if comments==True:
            print("Limit set to {}\n".format(limit))
        for n in tqdm(range(len(datasets[:limit])), disable=tqmdis):
            dataset=datasets[n]
            if "SCOPE:NAME[DIDTYPE]" in dataset or "--------------------------------------------------------------------------------" in dataset:
                pass
            else:
                L=(os.popen("rucio list-file-replicas --rses {} {}".format(rse,dataset)).read
                    ()).split('\n')
                L=[file for file in L if rse in file]
                L=[file+str(dataset) for file in L]
                number_of_files_in_dataset[dataset]=len(L)
                list_of_files.extend(L)
    #Remove any empty datasets from the dataset number dictionary
    number_of_files_in_dataset = dict( [(k,v) for k,v in number_of_files_in_dataset.items() if
        not v==0])

    #Remove duplicate files from the file list
    list_of_files=list(set(list_of_files))

    #Turn each file entry into a list by splitting on |
    new_new_list=[file.replace(" ","").split("|") for file in list_of_files]
    #Removing the first two entries that don't contain important information
    new_new_list=[file[2:] for file in new_new_list]

    #for the rse add all file list to the dictionary
    datasets_rse[rse]=new_new_list

    #Print statistics for a rse
    if comments==True:
        print("\n")
        print("For the rse {} we have this many files:".format(rse))
        print(len(datasets_rse[rse]))
```

```
    #If no files were found for rse remove it from dictionary
    datasets_rse = dict( [(k,v) for k,v in datasets_rse.items() if len(v)>0])
    return(datasets_rse, number_of_files_in_dataset)

#clean the data by removing useless information and remove the filename from the storage path
def clean_up_datasets_rse(datasets_rse):
    for rse in datasets_rse:
        if "GRIDFTP" in rse:
            grid_settings=open("config.txt","r")
            grid_settings_list=grid_settings.readlines()
            for line in grid_settings_list:
                if rse in line:

                    address_to_change=line.split(",")[1]
                    change_with=line.split(',')[2]
                    change_with=change_with.replace('\n',"")

            if comments==True:
                print("\nCleaning up data about files at {}".format(rse))

            dataset_list=datasets_rse[rse]
            #print(dataset_list[:2])
            for n in tqdm(range(len(dataset_list)), disable=tqmdis):
                dataset=dataset_list[n]

                dataset[3]=dataset[3].replace(dataset[0],"")

                dataset[3]=dataset[3].replace(address_to_change,change_with)



        else:
            if comments==True:
                print("\nCleaning up data about files at {}".format(rse))

            dataset_list=datasets_rse[rse]
            #print(dataset_list[10:])
            for n in tqdm(range(len(dataset_list)), disable=tqmdis):
                dataset=dataset_list[n]
                dataset[3]=dataset[3].replace(dataset[0],"")
                dataset[3]=dataset[3].replace("{}:file://".format(rse),"")
    return (datasets_rse)


#For a provided file file2 and directory dir2 it will return the adler32 checksum in hexadecimal
def get_adler32_checksum(file2):
    BLOCKSIZE=256*1024*1024
    asum=1
    #If we don't want to compare the checksum and only see if the file exists we try to open the
        file. If we are able to open the file we set asum=0 and if we fail we set it to None
    if checksum==False:
        try:
            with open("{}".format(file2),"rb") as f:
                asum=0
            f.close()
        except:
            asum=None
    #if we want to do a checksum, we try to open the file. If we fail we set it to None, else we
        calculate the adler32 checksum for the file.
    else:
        try:
            with open("{}".format(file2),"rb") as f:
                while True:
                    data = f.read(BLOCKSIZE)
                    if not data:
                        break
                    asum = adler32(data, asum)
                    if asum < 0:
                        asum += 2**32
        except:
```

```
            asum=None
    return(asum)


#compares the adler34 checksum for the files we matched between storage and rucio. Outputs the
    rucio entries without a match, the checksum of the file in storage, the checksum as reported
    by rucio.
def compere_checksum(datasets_rse, number_of_files_in_dataset):
    #Get the current time. Used to name the folders
    now = datetime.now()
    now=str(now)
    now=now.replace(" "," ")
    for rse in datasets_rse:
        now=rse+"_"+now
        break
    if All==True:
        now="All"+"_"+now

    #Check if the Output folder exist, if not create it
    if not os.path.exists('output'):
            print("Output folders missing, creating a new one.")
            os.makedirs('output')
    if not os.path.exists('output/{}'.format(now)):
            os.makedirs('output/{}'.format(now))

    #The location for saving the output files
    save_dir='output/{}'.format(now)
    #Address to save the output for the files that failed the adler32 comparison
    failed_adles32=save_dir+"/"+"adler32_fail.txt"
    #Address to save the output for the files that exist in Rucio but are not found in storage
    not_found_addres=save_dir+"/"+"files_missing_storage.txt"
    #Address to save the output for the files that exist in Rucio and are also in storage
    found_addres=save_dir+"/"+"files_found_storage.txt"
    #Address to save the output for the files that exist in storage but not in Rucio
    missig_in_rucio_addres=save_dir+"/"+"files_missing_rucio.txt"
    #Address to save the output for the datasets and the number of files at a rse that exist in
        that dataset
    datasets_addres=save_dir+"/"+"datasets_and_numbers.txt"


    problem_dir_addres=save_dir+"/"+"problem_locations.txt"
    #A list of directories that are problematic
    problem_dir=[]

    for rse in datasets_rse:
        #list of all the directories that we found. Used to get all the files in storage.
        directory_list=[]

        #Dictionary of all the files that we found for a rse (why is this a dict not a list?)
        files_found={}
        #Dictionary of all the files that we did not find for a rse (why is this a dict not a list
            ?)
        files_not_found={}

        #Integer that is increased every time a file failed a checksum
        integ=0
        if comments==True:
            print("\nComparing the adler32 checksum in rucio with checksum in storage for rse {}."
                .format(rse))

        #get all the file at the rse
        datasets=datasets_rse[rse]

        #for each file in
        for n in tqdm(range(len(datasets)), disable=tqmdis):
            #get the information for each file
            file_data=datasets[n]
            #filename
            file=file_data[0]
```

```
        #the dataset the file belongs to
        dataset_file=file_data[-1]

        #The storage location for where the file is
        directory=file_data[3]
        #print(directory)
        #If the directory has not yet been encountered, add it to the list directory_list
        if directory not in directory_list:
            directory_list.append(directory)

        #Full path to the file
        fullpath=directory+file

        #The checksum that is registered in Rucio
        checksum_rucio=file_data[2]

        #Get the adler32 checksum for the file (or only check if it exist if checksum=False)
        checksum_dec=get_adler32_checksum(fullpath)

        #The output to be put in the file with files we did not find in storage
        not_found_str=str(file)+","+directory+","+dataset_file+"\n"

        #added endline so it works in the txt document
        #file=file+"\n"
        #If we had no checksum, if we found the file add it to the file_found dictionary
        if checksum_dec==0:
            if rse in files_found:
                files_found[rse].append(file)
            else:
                files_found[rse]=[]
                files_found[rse].append(file)
        elif checksum_dec==None:
            #If we did not find the file, add it to the dictionary files_not_found
            if rse in files_not_found:
                files_not_found[rse].append(not_found_str)
            else:
                files_not_found[rse]=[]
                files_not_found[rse].append(not_found_str)
        else:
            #If we did a checksum and found the file add it to the file_found dictionary
            if rse in files_found:
                files_found[rse].append(file)
            else:
                files_found[rse]=[]
                files_found[rse].append(file)
            #convert the decimal output from the adler32 checksum to a hexadecimal, so we can
                compare to Rucio
            checksum_hex=hex(checksum_dec)
            #clean the checksum
            checksum_hex=checksum_hex.lstrip("0x").rstrip("L")

            #If the checksums match do nothing, else increase the integer by one and write to
                the adler32 txt file
            if str(checksum_hex) in checksum_rucio:
                pass
            else:
                integ=integ+1
                os.system("echo␣{},{},{}␣>>␣{}".format(file,checksum_rucio,str(checksum_hex),
                    failed_adles32))
    #get the number of files that failed and succeeded the search in storage
    number_failed_files=0
    number_sucesfull_files=0
    for rse in files_not_found:
        number_failed_files=number_failed_files+len(files_not_found[rse])

    for rse in files_found:
        number_sucesfull_files=number_sucesfull_files+len(files_found[rse])

    #print out some information about the comparison
    if comments==True:
```

```
    print("\nFor␣the␣rse␣{}␣we␣found␣that␣{}␣files␣were␣missing␣in␣storage␣out␣of␣{}.".
        format(rse, number_failed_files, len(datasets)))
    if checksum==True:
        print("We␣found␣{}␣corrupted␣files␣out␣of␣{}".format(integ,(len(datasets))))

#Write information about the files we did not find in storage to a txt file
not_found= open("{}".format(not_found_addres),"w+")
for n in files_not_found:
    for k in files_not_found[n]:
        not_found.write(k)
not_found.close()

#Write information about the files we found in storage to a txt file
found= open("{}".format(found_addres),"w+")
for n in files_found:
    for k in files_found[n]:
        output=k+"\n"
        found.write(output)
found.close()



#A list for the files that exist in storage
files_in_storage=[]

#A list of the files that exist in storage but not in Rucio
files_not_in_Rucio=[]



#For each directory get all the files there for comparison with the files we found
for n in range(len(directory_list)):
    directory=directory_list[n]

    stuff_to_add=list(os.popen("ls␣{}␣||␣echo␣false".format(directory)))
    if ['false\n']==stuff_to_add:
        #If directory is not found/not available add to problem list
        problem_dir.append(directory)
    else:
        #clean the output
        stuff_to_add=[f.replace("\n","") for f in stuff_to_add]
        #add the directory so it's easier for comparison program and tur each element into
            a list
        stuff_to_add=[f+","+directory for f in stuff_to_add]
        stuff_to_add=[f.split(",") for f in stuff_to_add]

        files_in_storage.extend(stuff_to_add)




if comments==True:
    print("\nFor␣each␣storage␣location␣find␣what␣files␣in␣storage␣is␣not␣registered␣in␣
        Rucio.")

#check if the file in storage exists in the list of found files. If not, add that file to
    the list files_not_in_Rucio
for n in tqdm(range(len(files_in_storage)), disable=tqmdis):
    file=files_in_storage[n]
    if file[0] not in files_found[rse]:
        files_not_in_Rucio.append(file)

#Write the files ot in Rucio to a txt file
not_in_rucio=open("{}".format(missig_in_rucio_addres),"w+")
for file in files_not_in_Rucio:
    output=str(file[0])+","+str(file[1])+"\n"
    not_in_rucio.write(str(output))
not_in_rucio.close()

if comments==True:
```

```
        print("\nWe found {} files in storage that are not registered in Rucio. PS: Unless you
            runned a full search for all scopes and datasets this output does not mean much."
            .format(len(files_not_in_Rucio)))

#Write the datasets and the number of files in them to a txt file
datasets= open("{}".format(datasets_addres),"w+")
for dataset in number_of_files_in_dataset:
    output=str(dataset) +","+str(number_of_files_in_dataset[dataset])+"\n"
    datasets.write(output)
datasets.close()

#If problem files exist, write them to a txt file
if len(problem_dir)>0:
    prob_loc= open("{}".format(problem_dir_addres),"w+")
    for direct in problem_dir:
        output=str(direct)+"\n"
        prob_loc.write(output)
    prob_loc.close()
```

Listing 2: The code containing the functions for the comparison tool, written in Python.

```
import os
from zlib import adler32
from datetime import datetime
from tqdm import *
from DDS_comparison_tool import *
from numpy import size
import test2



if not os.path.exists('classifier'):
        os.makedirs('classifier')
if not os.path.exists('classifier/checked.txt'):
        os.mknod('classifier/checked.txt')

def get_runs():
    output_list=next(os.walk('output'))[1]

    checked=[]

    checked_file=open("{}".format('classifier/checked.txt'),"r+")
    checked_file_lines=checked_file.readlines()
    for line in checked_file_lines:
        checked.append(line)
    output_list_to_check=[]
    for file in output_list:
        if file not in checked:
            output_list_to_check.append(file)
    #output_list_to_check=[file for file in output_list if file not in checked]
    for file in output_list:
        output=file+"\n"
        checked_file.write(output)
    checked_file.close()
    return(output_list_to_check,checked)

def get_files_in_runs(output_list_file):
    files_in_output=next(os.walk('output/{}'.format(output_list_file)))[2]
    return(files_in_output)


def files_missing_storage_with_datasets(output_file):
    files_batch_not_a_problem=[]
    files_really_problem_batch=[]
    files_missing_batch=[]
    files_missing_rucio_list=[]



    files_missing_storage=open("output/{}/files_missing_storage.txt".format(output_file),"r")
    files_missing_storage_lines=files_missing_storage.readlines()
    files_missing_storage_list=[]



    datasets_and_numbers=open("output/{}/datasets_and_numbers.txt".format(output_file),"r")
    datasets_and_numbers_list=[]
    print("Extracting information about datasets.")
    for line in tqdm(datasets_and_numbers):

        datasets_and_numbers_list.append(line.split(","))
    datasets_and_numbers.close()

    print("Counting and comparing the number of files missing in storage and files registered to
        dataset.")
    for file in tqdm(datasets_and_numbers_list):
        batch=file[0]
        number=int(file[1])
        n=0
```

```
    for stuff in files_missing_storage_lines:
        if batch in stuff:
            n=n+1
    if n/number>0:
        if n/number>0.2 and not n/number==1:
            files_really_problem_batch.append([batch,n/number])
        elif n/number==1:
            files_missing_batch.append([batch,n/number])
        else:
            files_batch_not_a_problem.append([batch,n/number])
    else:
        files_batch_not_a_problem.append([batch,n/number])

print("Cleaning the output.")
files_really_problem_batch_dict={}
for file in files_really_problem_batch:

    file=file[0]
    for stuff in files_missing_storage_lines:
        if file in stuff:
            output1=stuff.replace("\n","").split(",")
            batch=output1[2]
            output=output1[:1]

            if batch in files_really_problem_batch_dict:
                files_really_problem_batch_dict[batch].append(output)
            else:
                files_really_problem_batch_dict[batch]=[]
                files_really_problem_batch_dict[batch].append(output)


files_missing_batch_dict={}
for file in files_missing_batch:

    file=file[0]
    for stuff in files_missing_storage_lines:
        if file in stuff:
            output1=stuff.replace("\n","").split(",")
            batch=output1[2]
            output=output1[:1]

            if batch in files_missing_batch_dict:
                files_missing_batch_dict[batch].append(output)
            else:
                files_missing_batch_dict[batch]=[]
                files_missing_batch_dict[batch].append(output)

files_batch_not_a_problem_list1=[]
for file in files_batch_not_a_problem:
    file=file[0]
    for stuff in files_missing_storage_lines:
        if file in stuff:
            output=stuff.replace("\n","").split(",")
            files_batch_not_a_problem_list1.append(output)
files_batch_not_a_problem_list=[]
problem_locations_list_files={}

if os.path.exists(("output/{}/problem_locations.txt".format(output_file))):
    problem_locations=open("output/{}/problem_locations.txt".format(output_file),"r")
    problem_locations_lines=problem_locations.readlines()


    for file in files_batch_not_a_problem_list1:
        directory=file[1]
        if directory+"\n" in problem_locations_lines:
            if directory in problem_locations_list_files:
                problem_locations_list_files[directory].append(file[0])
            else:
                problem_locations_list_files[directory]=[]
                problem_locations_list_files[directory].append(file[0])
```

```
        else:
            files_batch_not_a_problem_list.append(file)
else:
    files_batch_not_a_problem_list=files_batch_not_a_problem_list1


if not os.path.exists('classifier/{}'.format(output_file)):
    os.makedirs('classifier/{}'.format(output_file))

if not os.path.exists('classifier/{}/files_missing_storage'.format(output_file)):
    os.makedirs('classifier/{}/files_missing_storage'.format(output_file))

if len(files_really_problem_batch_dict)>0:
    if not os.path.exists('classifier/{}/files_missing_storage/batch_problem/'.format(
        output_file)):
        os.makedirs('classifier/{}/files_missing_storage/batch_problem/'.format(output_file))
if len(files_missing_batch_dict)>0:
    if not os.path.exists('classifier/{}/files_missing_storage/batch_missing//'.format(
        output_file)):
        os.makedirs('classifier/{}/files_missing_storage/batch_missing//'.format(output_file))
if len(problem_locations_list_files)>0:
    if not os.path.exists('classifier/{}/files_missing_storage/problem_locations_files//'.
        format(output_file)):
        os.makedirs('classifier/{}/files_missing_storage/problem_locations_files//'.format(
            output_file))

print("Writing output to files.")

if len(files_really_problem_batch_dict)>0:
    for batch in files_really_problem_batch_dict:
        files_really_problem_file=open('classifier/{}/files_missing_storage/batch_problem/{}.
            txt'.format(output_file,batch),"w+")
        for file in files_really_problem_batch_dict[batch]:
            output=file[0]+"\n"
            files_really_problem_file.write(output)
        files_really_problem_file.close()


if len(problem_locations_list_files)>0:
    for direct in problem_locations_list_files:
        problem_locations_list_files_file=open('classifier/{}/files_missing_storage/
            problem_locations_files/problem_locations_files.txt'.format(output_file,direct),"w
            +")
        output_1=""
        for file in problem_locations_list_files[direct]:
            output_1=output_1+file
        output=direct+","+output_1+"\n"
        problem_locations_list_files_file.write(output)
        problem_locations_list_files_file.close()


if len(files_missing_batch_dict)>0:
    for batch in files_missing_batch_dict:
        files_missing_batch_file=open('classifier/{}/files_missing_storage/batch_missing/{}.
            txt'.format(output_file,batch),"w+")
        for file in files_missing_batch_dict[batch]:
            output=file[0]+"\n"
            files_missing_batch_file.write(output)
        files_missing_batch_file.close()

if len(files_batch_not_a_problem_list)>0:
    files_other_problem_file=open('classifier/{}/files_missing_storage/other_problem.txt'.
        format(output_file),"w+")
    for file in files_batch_not_a_problem_list:
        output=file[0]+","+file[1]+"\n"
        files_other_problem_file.write(output)
    files_other_problem_file.close()

#make the messages
for stuff in files_missing_batch:
```

```
            file=stuff[0]
            procentage=stuff[1]
            output=("The batch {} is present in Rucio but is missing from storage.".format(file))
            summery_problems.append(output)

        for stuff in files_really_problem_batch:
            file=stuff[0]
            procentage=stuff[1]*100
            output=("The batch {} in Rucio migh have some problem, becouse {} % of it is missing in
                storage.".format(file,procentage))
            summery_problems.append(output)


def files_missing_rucio(output_file):
    #get all the files tat exist in storage but not in Rucio
    files_missing_rucio=open("output/{}/files_missing_rucio.txt".format(output_file),"r")

    files_missing_rucio_lines=files_missing_rucio.readlines()

    files_found_storage=open("output/{}/files_found_storage.txt".format(output_file),"r")
    files_found_storage_list=files_found_storage.readlines()




    runs={}
    for files in files_found_storage_list:
        file=files.replace("\n","")
        #file=files.split(",")

        filename_list=file[:file.rindex("_")]

        filename=filename_list[:filename_list.rindex("_")]
        filename=filename.replace(" ","")
        if not filename in runs:
            runs[filename]=1
        else:
            runs[filename]=runs[filename]+1

    files_missing_rucio_files=[]
    print('Cleaning the missing from Rucio data')
    for n in tqdm(range(len(files_missing_rucio_lines))):
        line=files_missing_rucio_lines[n]
        line=line.replace("[","")
        line=line.replace("'","")
        line=line.split("\n")
        line=[item.split(",") for item in line]
        line=[item for item in line if len(item)>1]
        files_missing_rucio_files.extend(line)

    files_missing_rucio_files_dict={}

    print('Looking for duplicates among missing files and extraxting file name')
    for n in tqdm(range(len(files_missing_rucio_files))):
        file=files_missing_rucio_files[n]
        filename=file[0]
        address=file[1]

        filename_list=filename[:filename.rindex("_")+1]

        if filename_list in files_missing_rucio_files_dict:
            files_missing_rucio_files_dict[filename_list].append(filename)
        else:
            files_missing_rucio_files_dict[filename_list]=[]
            files_missing_rucio_files_dict[filename_list].append(address)
            files_missing_rucio_files_dict[filename_list].append(filename)


    files_missing_rucio_files_dict2={}
    many=[]
```

```
few=[]
print("Search␣for␣duplicates␣among␣the␣files␣in␣Rucio␣and␣in␣storage")
for n in tqdm(files_missing_rucio_files_dict):

    k=[file for file in files_found_storage_list if n in file]

    files_missing_rucio=files_missing_rucio_files_dict[n]

    files_missing_rucio.extend(k)
    if len(k)>0:
        many.append(files_missing_rucio)
    else:
        few.append(files_missing_rucio)
print("Among␣the␣files␣missing␣from␣Rucio␣{}␣were␣duplicates␣{}␣are␣regular␣missing␣files".
    format(len(many),len(few)))



adler_no_problem=[]
adler_problem=[]

print("Calculate␣checksum␣for␣duplicates␣to␣se␣if␣they␣are␣corrupted␣or␣just␣a␣copy.")
for n in tqdm(range(len(many))):
    file=many[n]
    addres=file[0].replace("␣","")
    adler32=[]
    for filename in file[1:]:
        filename.replace("␣","")
        fulladress=addres+filename
        fulladress=fulladress.replace("␣","")
        adler32output=get_adler32_checksum(fulladress)
        adler32.append(adler32output)

    adler32=list(set(adler32))
    if len(adler32)>1:
        adler_problem.append(file[1:])
    else:
        adler_no_problem.append(file[1:])
print("Number␣of␣currupted␣files")
print(len(adler_problem))



problem_runs={}
for file in adler_problem:
    for filename in file:
        filename=filename[:filename.rindex("_")]
        filename_name=filename[:filename.rindex("_")]
        filename_name=filename_name.replace("␣","")
        if filename_name not in problem_runs:
            problem_runs[filename_name]=0
        else:
            problem_runs[filename_name]=problem_runs[filename_name]+1


problem_runs_2=[]
missing_runs_2=[]
no_problem_runs_2=[]
for filename in problem_runs:
    number_problem_files=problem_runs[filename]
    if number_problem_files==0:
        no_problem_runs_2.append(filename+","+str(0))
    else:
        number_all=runs[filename]

        frac=number_problem_files/number_all

        if frac>0.2 and not frac==1:
            problem_runs_2.append(filename+","+str(frac))
```

```
        elif frac==1:
            missing_runs_2.append(filename+","+str(frac))
        else:
            no_problem_runs_2.append(filename+","+str(frac))
print("runs with problem")
print(len(problem_runs_2))
print("runs missing problem")
print(len(missing_runs_2))
print("runs no problem")
print(len(no_problem_runs_2))


if not os.path.exists('classifier/{}'.format(output_file)):
    os.makedirs('classifier/{}'.format(output_file))

if not os.path.exists('classifier/{}/files_missing_rucio'.format(output_file)):
    os.makedirs('classifier/{}/files_missing_rucio'.format(output_file))

if len(adler_problem)>0:
    currupted_files=open('classifier/{}/files_missing_rucio/currupted_duplicate.txt'.format(
        output_file),"w+")
    for file in adler_problem:
        output=""
        for name in file:
            output=output+","+name
        output=output+"\n"
        currupted_files.write(output)
    currupted_files.close()

if len(adler_no_problem)>0:
    regular_files=open('classifier/{}/files_missing_rucio/regular_duplicate.txt'.format(
        output_file),"w+")
    for file in adler_no_problem:
        output=""
        for name in file:
            output=output+","+name
        output=output+"\n"
        regular_files.write(output)
    regular_files.close()

if len(problem_runs_2)>0:
    runs_with_problem_file=open('classifier/{}/files_missing_rucio/runs_with_problem.txt'.
        format(output_file),"w+")
    for file in problem_runs_2:
        output=file+"\n"
        runs_with_problem_file.write(output)
    runs_with_problem_file.close()

if len(missing_runs_2)>0:
    missing_runs_files=open('classifier/{}/files_missing_rucio/missing_runs_2.txt'.format(
        output_file),"w+")
    for file in missing_runs_2:
        output=file+"\n"
        missing_runs_files.write(output)
    missing_runs_files.close()

if len(few)>0:
    missing_files=open('classifier/{}/files_missing_rucio/missing.txt'.format(output_file),"w+
        ")
    #print((few))
    for file_out in few:
        output=str(file_out)+"\n"
        missing_files.write(output)
    missing_files.close()


if len(many)>0:
    missing_files=open('classifier/{}/files_missing_rucio/many_missing.txt'.format(output_file
        ),"w+")
    #print((few))
```

```
        for file_out in many:
            output=str(file_out)+"\n"
            missing_files.write(output)
        missing_files.close()

    for stuff in problem_runs_2:
        stuufer=stuff.split(",")
        file=stuufer[0]
        procentage=str((float(stuufer[1])*100))
        output=("The run {} in Rucio migh have some problem, becouse {} % of the files missing in
            Rucio are duplicates.".format(file,procentage))
        summery_problems.append(output)

    for stuff in missing_runs_2:
        stuufer=stuff.split(",")
        file=stuufer[0]
        procentage=str((float(stuufer[1])*100))
        output=("The run {} in Rucio probably has some problem, becouse {} % of the files missing
            in Rucio are duplicates.".format(file,procentage))
        summery_problems.append(output)

def adler32fail(output_file):
    adler32failfile=open("output/{}/adler32_fail.txt".format(output_file),"r")
    adler32failfile_lines=adler32failfile.readlines()
    for line in adler32failfile_lines:
        lines=line.split(",")
        if ['\n']==lines:
            pass
        else:
            filename=lines[0]
            rucio=lines[1]
            storage=lines[2].replace("\n","")
            output=("The file {} has been corrupted. The value for the Adler32 checksum in Rucio
                is {} but the checksum in storage is {}.".format(filename,rucio,storage))
            summery_problems.append(output)

def runner(output_list_to_check,files_in_output):
    if "files_missing_storage.txt" in files_in_output:
        if "datasets_and_numbers.txt" in files_in_output:
            print("\nClassifying files present in Rucio but missing from storage.\n")
            files_missing_storage_with_datasets(output_list_to_check)
    if "files_missing_rucio.txt" in files_in_output:
        print("\nClassifying files present in storage but missing from Rucio.\n")
        files_missing_rucio(output_list_to_check)
    if "adler32_fail.txt" in files_in_output:
        adler32fail(output_list_to_check)
    if len(summery_problems)>0:
        print("\nA summery of the problems found\n")
        summery_problems_file=open('classifier/{}/summery_problems.txt'.format(
            output_list_to_check),"w+")
        for p in summery_problems:
            print(p)
            output=str(p)+"\n"
            summery_problems_file.write(output)
        summery_problems_file.close()



output_list_to_check=get_runs()[0]
for k in range(len(output_list_to_check)):
    if "All" not in output_list_to_check[k]:
        pass
    else:
        summery_problems=[]
        output_file=get_files_in_runs(output_list_to_check[k])

        runner(output_list_to_check[k],output_file)
```

Listing 3: The code for the classification tool, written in Python.