SciFi - Increasing Database Adaptation in Data Science with an Embedded Asset Store

Annett Ungethüm[1], Martin Poppinga[2], Katharina Kögel[3] and Matthias Rarey[4]

[1]*Universität Hamburg, Center for Data and Computing in Natural Sciences, Germany*
[2]*Universität Hamburg, Department for Informatics, Databases and Information Systems, Germany*
[3]*Universität Hamburg, Germany*
[4]*Universität Hamburg, ZBH - Center for Bioinformatics, Germany*

Abstract

During the complex discovery processes in data science, multiple kinds of data have to be managed, e.g. experimental data, metadata, and computationally produced data. Managing this data has become a bottleneck limiting the potential rate of discoveries. On the one side, creating individual solutions to handle the data produces an overhead on the scientists time. On the other hand, existing database management systems which could reduce the development time while offering sophisticated optimization and execution engines, are often not used for a variety of reasons. For instance, the initial training to use such systems takes time, which is a rare good amongst scientists. Further, even the existing systems and repositories cannot completely fulfill the needs for data management in science while providing the necessary convenience. To simplify the process of storing, accessing, and sharing data, while providing reasonable performance, digital asset management (DAM) systems provide solid solutions. While DAM is already common in other fields, such as photography or music, it is hardly used in science, mainly due to a lack of available and applicable systems. To close this gap, we present ScienceFiles (*SciFi*), an embedded asset store specially developed for scientific data. *SciFi* consists of an extensible framework, an extensible shell which serves as a stand-alone asset store, and an extension for PyTorch to make data directly available for machine learning. To ensure the usability of our solution, we chose a lightweight design that runs on laptops and lab PCs without requiring special permissions or regular administration.

1. Introduction

Discovery processes in science incorporate an increasing amount of data from high resolution and high throughput instruments, intermediates and results of computational analyses, and from a growing number of collaborative work. The management and access of this data has become a bottleneck in data analysis pipelines. In a community of system engineers and in large-scale applications, databases are commonly used to manage huge amounts of data. However, most data scientists are not system engineers and do not work with large-scale applications. The focus of data scientists is mostly on the methods and the used toolkits or hand-crafted scripts, not on the management of the used and produced data. Although, they could benefit from the optimized processing of analytical queries in modern database management systems. The applications themselves are often highly specialized but small pieces of software which are

LWDA'24: Lernen, Wissen, Daten, Analysen. September 23–25, 2024, Würzburg, Germany
annett.ungethuem@uni-hamburg.de (A. Ungethüm); martin.poppinga@uni-hamburg.de (M. Poppinga); katharina.koegel@studium.uni-hamburg.de (K. Kögel); matthias.rarey@uni-hamburg.de (M. Rarey)

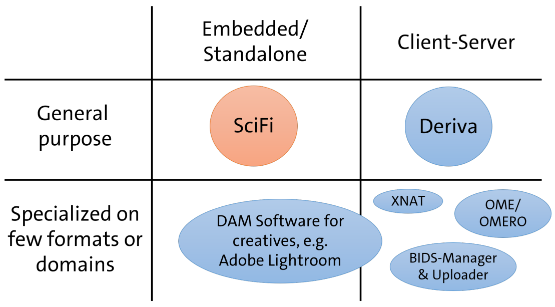


Figure 1: Landscape of systems for digital asset management (DAM) in Science

only developed and used by a small group of researchers. One reason for this is the fear of being scooped. Thus, applications are not always published and not evaluated by external scientists. Another reason is the limited range of potential users for niche research fields. Limited privileges and restricted computing times on HPC clusters are an additional obstacle to migrating these applications to a larger system. As a result of this situation, the most popular computing platforms for data science in 2021, when this question was last asked in the Kaggle survey, were still laptops and personal computers [1]. The same review one year later in 2022, which is the most current available version, shows more interesting aspects, e.g. that a third of the respondents did not spent any money in the cloud. Further, 18.4% of the respondents do not use any data storage and another 22.3% use SQLite which is not optimized for analytical queries. Moreover, SQL has decreased in popularity while the popularity of Python has increased since 2021. These developments show that data systems are either not used at all or not in an efficient way.

Raasveldt et al. already showed that current solutions are not equipped for analytical workloads on local personal computers [2]. As a consequence, they developed DuckDB, an embedded RDBMS for analytical workloads. However, not all data is relational, and thinking in relations is not intuitive for every data collection. Furthermore, to understand data, metadata is necessary while there is no universal way of how metadata is stored, processed, and presented. In the worst case, it is implicit knowledge of the scientist. In the best case, it is organized and digital, e.g. in an excel sheet but only rarely in a relational database. Even if it is organized, the connection to the data has to be done manually, e.g. for finding data which fulfills certain requirements. This is often done by hand-crafted scripts which have to be rewritten for every new dataset. This is against the concept of FAIR (findable, accessible, interoperable, reusable) data sharing principles [3, 4].

Public repositories of scientific data sometimes offer additional functionality or even a whole processing pipeline, e.g. the Protein Data Bank[1] [5]. However, these are curated repositories of highly domain-specific data. They are not systems which can be used with any data and instantiated at any location. A class of systems which is able to fulfill this, is digital asset management (DAM) systems.

DAM systems are used for organizing and querying data and metadata. Popular DAM applications like Adobe Lightroom, Google Photos, or Apple Photos are widely used and provide a local version as well as a server-based solution. This way, asset management has become

[1]RCSB.org, Accessed: 2023-12-12

a tool and not a task for creatives, just like it should be a tool for scientists rather than time consuming additional work. In a scientific context, the mentioned systems could be used for managing images that are the result of experiments or computational methods, but not for managing the according metadata. While metadata for a photographer is the information stored in the exif tags, scientific metadata can include an arbitrary number of additional entries, e.g. environment variables, textual descriptions, and responsible scientists. Accordingly, the queries on metadata also look different. While a photographer might filter "all images with a 5-star rating", a simple query of a scientist will rather be in the fashion of "Which structure has a size smaller than x and was observed in environment y?" to name just an example. Such queries are prime examples for analytical queries. In science, DAM systems are mostly domain-specific like BIDS-manager for electrophysiology and neuroimaging [6]. This way, the necessary and optional fields for metadata entries can be predefined. In contrast to these systems, DERIVA [7] is made for general purpose usage. However, it is designed as a collection of web services. Thus, it is not an ideal solution for local processing on a personal computer. Figure 1 shows an overview of the existing DAM environment, including some examples. There is a clear gap for serverless general purpose scientific asset management.

In this work, we aim to close this gap by introducing *SciFi*, an embedded asset store for scientific data. To encourage the use of our solution, we also build an integration into PyTorch, one of the most popular frameworks for machine learning. In detail, our contributions are the following:

- First, we give a general overview of the requirements of scientific asset management and explain which systems we consider to incorporate into our solution in Section 2.
- Then, we present our architecture in Section 3.
- In Section 4 we present an evaluation using real world datasets.
- We discuss related work in Section 5.
- Finally, we give a brief outlook and conclusions in Section 6 and Section 7.

2. Background

There is a vast variety of reasons why data systems are not used by all scientists working with data. They range from missing privileges on the used systems to a lack of resources to administrate a local server setup, or the time to learn a new query language. Additionally, the specific processes and formats differ between domains and even between individual use-cases. However, according to our experience, there are common requirements across all domains:

- **Simplicity.** It must be easy to get existing data into the asset store. This means, that the user should not be required to define a schema for the assets or generally do anything that is more complicated than just copying data to another file.
- **Fast retrieval of assets.** Ideally, retrieving an asset via the DAM system should be faster than opening, reading, and returning the contents of a file on disc. This is especially crucial if a large number of files is processed.
- **Efficient usage of disc space and the file system when storing assets.** Using a flat file system to store large numbers of files increases the access time to these files and can even make the file system unusable. Further, when storing files smaller than the cluster

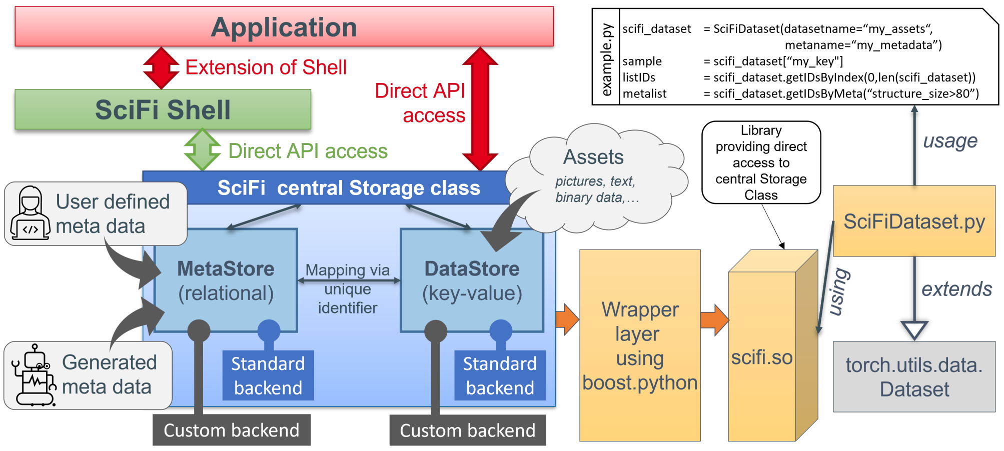
size of the disc, the remaining space in the cluster is not usable to any other file. Thus, the combination of many and small files results in slow access on data which blocks more space than necessary. Although disc space is relatively cheap, lengthy processes for material procurement, especially in publicly funded research institutions, can still produce a lack of available storage space.

- **Archiving data.** Discs dedicated to archiving data usually have a large capacity, but also large cluster sizes. This makes the previously mentioned issue of storing many small files more severe.
- **Storing, scanning, and analyzing metadata with reasonable performance.** Metadata is usually structured data, often organized in tables. Despite there being powerful systems which are specialized on fast processing of such data, the hand-crafted python script or excel are common tools to be used.
- **Connection between assets and metadata.** It must be possible to retrieve not only the metadata for a given asset, but also assets depending on their metadata or filters on their metadata.
- **Export of datasets for sharing data to ensure reproducibility.** This can be reached by using free and open-source formats, and systems which support these formats.

Based on these requirements, we built our DAM system, *SciFi*, on top of two different kinds of database systems: 1) For storing the metadata, we use a relational database system which offers a powerful query optimizer and high performance for analytical tasks. 2) For storing the assets, we use a key-value store (KVS), which offers limited functionality but can ingest every type of document. It is able to return assets with reasonable performance, mostly independent of the number of stored assets. This offers some advantages over storing the assets in a blob type column in a relational database. For instance, the data and metadata stay separated and can be exchanged easily and fast, e.g. if the format of the data changes. Another benefit is the optimization for fast writing and returning of values in a key-value-store. This concept of combining a KVS and a relational database is close to DERIVA [7] which, amongst other layers, uses PostgreSQL to store and query structured data, and an object store which holds arbitrary byte sequences. In contrast to DERIVA, our solution is supposed to run on laptops and personal computers without any overhead for administration. Thus, we only consider embedded systems. Another required features is that these systems must be open-source as requested by a number of scientists.

3. SciFi Architecture

The key element of *SciFi* is the combination of an embedded key-value store and an relational database to create a serverless asset store. While we aim to provide an easily applicable standalone solution, our approach should also be extensible and customizable to the needs of individual user groups. For this reason, we developed an extensible C++ framework and an application on top of it. For making data stored with *SciFi* directly usable for machine learning, we also extended the *Dataset* class of PyTorch.



```
scifi_dataset   = SciFiDataset(datasetname="my_assets",
                               metaname="my_metadata")
sample          = scifi_dataset["my_key"]
listIDs         = scifi_dataset.getIDsByIndex(0,len(scifi_dataset))
metalist        = scifi_dataset.getIDsByMeta("structure_size>80")
```

Figure 2: Overview of *SciFi*. Metadata and file contents are stored by different engines and in a different format to ensure compatibility with all file types while enabling fast analysis of metadata. Alternative backends can easily be plugged in and the shell is extensible according to the needs of the users.

3.1. Architecture overview

Figure 2 shows an overview of our *SciFi* architecture. It consists of four main parts:

SciFi framework (blue): Our framework incorporates two different storage interfaces, one for the assets (*DataStore*) and one for the metadata (*MetaStore*). The access to the two storages, i.e. the API, is provided by a central storage class. The *DataStore* is an interface for a key-value store (KVS) while the *MetaStore* is an interface for a relational database. Hence, a backend for each storage is required which implements system specific functions, such as the get, put, and delete functions of the KVS. We provide standard backends. However, a custom backend (grey) can easily be plugged in by providing the according template parameter when instantiating the central storage class. Since the *DataStore* and the *MetaStore* do not communicate directly with each other, the backends for them are independent of each other. The identification of assets and their metadata is done by a unique identifier. In the *DataStore*, unique identifiers are the keys of the key-value pairs. In the *MetaStore*, the unique identifier is stored as the primary key. The central storage class is responsible for forwarding the queries to retrieve the assets and metadata with a matching key.

SciFi shell (green): The *SciFi shell* is an extensible application prototype providing basic functions, e.g. scanning a directory for assets or writing queried assets and metadata to the file system. Custom functions can be added to the shell by calling the `register_function` method and providing the function pointer, a command string, and a help string as parameters. Once all functions are added or if the predefined functions are sufficient, the `run` method is used to start the shell.

Application (red): While the *SciFi shell* already runs as a standalone application, it can be extended as explained above. Alternatively, the *SciFi* API may be accessed directly. This is useful when there is already existing code written by a scientist which should now run on data stored by SciFi. In this case, the contents of queried assets can be provided directly as a return value instead of being written to disc, which accelerates the access to this data significantly.

PyTorch integration (yellow): There is a number of different AI frameworks and libraries used in data science with PyTorch being one of the most popular. PyTorch is a framework for machine learning (ML). For enabling the users of *SciFi* to directly use their data in ML applications without having to write it back to the file system, we extended the PyTorch framework. In a first step, we wrapped the access to the *DataStore* and the *MetaStore* using *boost.python* to create a python library (scifi.so). Then, we extended the *Dataset* class of Pytorch using this library. This way, data in the *DataStore* can be accessed in a map-style manner. Additionally, the access to the *MetaStore* is used to determine the number of available assets as well as for filtering assets by their metadata. The figure shows example code of how this access works in practice.

3.2. The storages

The *MetaStore* and the *DataStore* are implemented as interface classes. In this context, a backend is a specialization of these classes which uses an RDBMS resp. a KVS to provide the system specific access methods to the data. For a serverless asset store, we can only consider embedded RDBMS and KVS to create our default backends.

DataStore: The DataStore should be able to store random data, independently of the format. For this reason, we chose to store the assets in a KVS. A backend for the DataStore only has to implement a small number of system-specific functions. The functions `insert`, `remove`, and `getSingle` wrap the `put`, `delete`, and `get` commands provided by every KVS. Additionally, an `open` function is required which opens the database and sets the initial options if required. Finally, for being able to create and read portable files containing all asset data, `import` and `writePortable` have to be implemented. `WritePortable` creates a single data file from the contents of a given set of files. This data file is persistent and self-contained, i.e. independent of any log files or structures in main memory. `Import` ingests an existing self-contained file into the asset store. All system-independent code is already implemented in the DataStore interface, e.g. scanning of directories or creation of the keys.

Unlike relational databases, key-value stores are often designed as embedded systems. Some examples are LevelDB, RocksDB, embedded Redis, and BerkeleyDB. We decided on RocksDB[2]for the standard backend, which is built on the code basis of LevelDB[3], but with a focus on being scalable to larger systems. As even conventional laptops are equipped with an ever growing number of cores and larger memory, and with the rising availability of fast remote storage, this is an important design factor. For instance, RocksDB uses parallelized I/O operations wherever possible. Additionally, it provides the possibility to write sorted data directly into a Sorted String Table without requiring intermediate data like skiplists or write ahead log files. This is crucial to generate portable data files.

MetaStore: The *MetaStore* holds two different kinds of metadata: automatically generated data, e.g. file extensions, and user-defined data, which can be anything as long as it is organized in a table. To spare the users learning SQL for filtering their data, they only have to provide their constraints, i.e. the WHERE-clause. The remainder of the query is automatically created. While this restricts the possible expressivity of the statement, it is sufficient for most use-cases and can act as an efficient prefilter for more complex scenarios.

[2]rocksdb.org, Accessed: 2024-07-14

[3]https://github.com/google/leveldb, Accessed: 2024-07-14

The functions required by SciFi are limited, but each potential backend system has its own way of iterating and representing query results, e.g. inserting special characters as delimiters or returning different string formats. This means that each backend has to implement the treatment of results individually. It is crucial that they are in a uniform format when they are passed to the DataStore via the central storage class. In detail, the following functions are required for a MetaStore backend: 1) `open` creates a new database or opens an existing database, 2) `execQuery`, `getResultAsString`, and `printResult` executes a SQL query, returns the result of the last query as string, and prints the result, 3) `getIDsByConstraint` takes a filter argument for the metadata, i.e. the WHERE-clause of a SQL query, constructs the complete SQL query, and returns all results. The interface implements more functions, e.g. to write the metadata of an asset into a file. However, they rely on the functions implemented in the backend.

In *SciFi*, we expect more analytical queries than transactions. For instance, changing or adding data is only done whenever an experiment or a data producing script has finished. The time-consuming part is the analysis of this data, e.g filtering it for certain constraints. Hence, we will use a system designed for online analytical processing (OLAP). The list of embedded, free, and open source OLAP systems is short but existing. It contains DuckDB [2] and the embedded versions of MonetDB, MonetDB/e and MonetDBLite [8]. We decided on DuckDB [2] as our standard backend, which is a column-oriented RDBMS offering an in-memory mode and a persistency mode. While it uses the shell and test cases of the more popular SQLite, its implementation relies on more analytics-optimized and state-of-the-art methods, e.g. a vectorized execution engine, and an optimizer which unnests subqueries before creating the logical query execution plan.

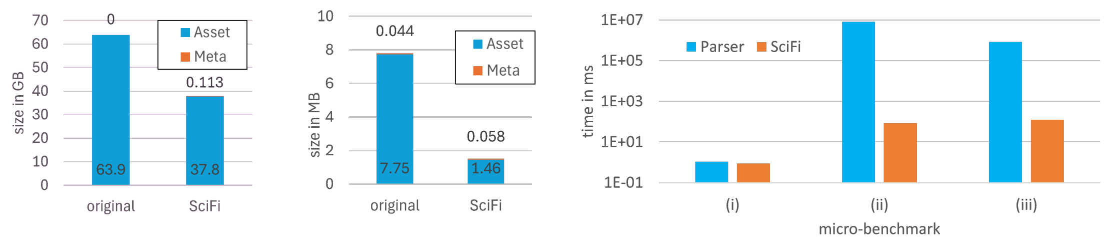
Meta Data Schema The first table holding user-defined metadata is always created automatically. Each additional table requires the user to provide a foreign key to this first table. This effectively enforces a Star Schema. For the automatically retrieved meta data, four tables are created by *SciFi*:

- *filedata* contains the unique key of each asset, whether this asset is compressed, the original path to the file, and the file extension.
- *metainfo* has a record for each relation created by the user. It contains the foreign key that connects the respective relation to the central *metadata* table.
- *filter* A user can store their frequently used filters. These are realized as views on the *MetaStore* side.
- *metadata* The fact table of the user-defined meta data. This table is created automatically but can be redefined by the user.

While some of the stored information is redundant because it also exists in the tables of the information schema of a relational database system, the access and usefulness of these internal tables can vary between different systems. Hence, the redundancy is a design decision to keep the number of required functions in the backend low.

3.3. API

To use the API, the central storage class must be instantiated. If no custom backends are provided as template parameters, the default backends are used. The member functions of the



a) Protein Data Bank[5] b) Radio Galaxies [9] c) Performance

Figure 3: Used disc space and performance. **a) and b)** The used disc space in the original format and using *SciFi*. For very small files, we can clearly see the offset of unused disc cluster space. The Protein Data Bank (a) encodes data and metadata in the same file. Thus, we did not distinguish between the size of the assets and the meta data. For the Radio Galaxy dataset (b), the meta data in *SciFi* is slightly larger because of the additional automatically created meta data. **c)** The performance of *SciFi* compared to a conventional approach. Note that the scale is logarithmic and that the data for the parser approach was reduced in (ii) and (iii).

created instance can now be used to work with SciFi. Names for existing persisted data of the MetaStore and the DataStore can be provided as parameters. If the names do not exist, a new store is created. In detail, the main functions provided by the API are the following: a) scanning directories for assets, b) adding metadata manually or from a csv file, c) return assets and metadata depending on filters on the metadata, d) return assets by their unique IDs, e) load metadata from a remote server. Assets and metadata can be written to files or returned as strings for direct further processing. A number of parameters can be used to ensure the exact desired behavior of each API call. Especially the function to scan a directory is highly parameterized. For instance, the default to create a key is to take the file name. Since this is not always what is used in the table containing the user-defined metadata, the creation of the keys can be modified, e.g. by removing prefixes and the file extension, or including a part of the directory hierarchy in the key.

Files can also be written to a temporary file system in main memory. The target directory, and if it points to disc or to main memory, is defined when instantiating the central storage. A symlink for convenient access is automatically created. Writing to a temporary file system has a number of advantages. First, it can improve the performance for other tools using the assets without having to change the access method, especially if the attached hard drive is on the slower side of the spectrum. Second, the risk to spam the disc with intermediates or copies, which will never be used again, is low. In some cases, copies of files are made for providing data in a folder structure appropriate for a certain tool. If this data is copied to a temporary file system, it is deleted at the next reboot. Only the original base data is left in the asset store. Hence, no data is lost, but there is also no redundant data.

4. Experimental Evaluation

We conducted our experimental evaluation on a local setup because we designed our system for being used locally. Our test system is a notebook equipped with an Intel i5-1145G7 CPU, which runs at a base frequency of 2.6 Hz and a peak frequency of 4.4 GHz. It features 4 physical cores and two threads per core. There are 8 GB of DDR4 main memory. The L1 data cache is 128 KiB, the L2 cache is 5MiB, and the L3 cache is 8 MiB. It runs on Ubuntu 22.04 LTS.

4.1. Data Sources and Data Size

Our test data originates from two public data sources. The first data source is the Protein Data Bank (PDB)[5]. The PDB[5] is a dataset of experimentally derived 3d structures of biological molecules. With over 60,000 structural biologists and data depositors having contributed more than 200,000 structures[4], the PDB is a corner stone of structural biology. It serves as input for other widely-used projects, such as AlphaFold[10]. Each structure is uniquely identified by its PDB ID, which is a 4 character code, and stored in an individual plain text file. This file contains meta data as well as the experimentally derived data, e.g. coordinates. We used an established parser for PDB files[5] to retrieve the metadata, which we then used to create the user-defined part of the *MetaStore* via the API. To create the *DataStore*, we could directly use the API without additional steps.

The second source is a set of radio galaxy images with curated labels[9]. The images are provided as .png files. The version we used contained 2158 different galaxy images. The metadata is provided as 3 separate csv files. We could use the API of *SciFi* to create the *MetaStore* and the *DataStore* without any intermediate steps.

Figure 3(a) and (b) show the required disc space of the original data and the same data in *SciFi*. The PDB dataset is available in different formats, all of them being one plain text file per PDB ID. The figure shows the legacy PDB format which produces smaller file sizes than the more recent mmcif format. However, both formats are widely used in research and existing pipelines depend on the data being in the exact format that is expected. Thus, providing the whole dataset in a relational format only, would break these pipelines. Since both datasets use an individual small file for each asset, the space cannot be used efficiently because each file takes up a whole disc cluster (block), even if the content is smaller than the cluster. This issue is efficiently solved by using a key-value store as the backend of the *DataStore*. The meta data only requires a small fraction of the overall size. However, the original PDB files encode data and meta data within the same file. Thus, we do not distinguish the sizes for the assets and the meta data in the original PDB data.

4.2. Performance

We chose the PDB dataset to show the performance benefits of *SciFi* over other approaches, because the size of the dataset exceeds the main memory size of our system. Our comparison is between *SciFi* and the use of the previously mentioned PDB parser.

We conducted the following micro benchmarks:

(i) Load an asset identified by its PDB ID into main memory. The parser is not needed for this benchmark because the file path can be reconstructed if the PDB ID is known.

(ii) Find all entries which fulfills the following criteria: country of citation must be 'UK'.

(iii) Find all entries which fulfill two criteria and require a join in our *MetaStore*, i.e. the country of citation must be 'UK' and the the database status must be 'released'. In the PDB file, the

[4]https://www.rcsb.org/pages/about-us/index, accessed 2023-12-13

[5]https://github.com/PDB-REDO/libcifpp, accessed 2023-12-13

latter is encoded as '_pdbx_database_status.status_code="REL"'.

For the parser, we selected only the first half of the dataset for our micro-benchmark (ii) to reduce the query runtimes in the conventional approach to a tolerable level, while *SciFi*uses the whole dataset. This also means that not all results are found. Even with this reduction, the constant thermic stress of accessing an internal disk constantly, caused bitflips on our hard drive. We did not experience this issue when creating our *DataStore*since we were not using an integrated hard drive for this purpose. While most of the damage could be reversed, the folder containing the PDB data was lost and we had to download it again. For this reason, we used only 10 GB of the original dataset for our micro-benchmark (iii). *SciFi*was using the whole dataset in all micro-benchmarks. Thus, our first and probably most important observation is that *SciFi* enables the user to work with datasets which would be too large to work with using a conventional approach on consumer-grade hardware.

Figure 3(c) shows the results of our micro-benchmarks. We use a logarithmic scale because the differences in (i) and (ii) are too large to show the results for both approaches on the same linear scale. The results show the mean of 5 runs. The first benchmark shows that there is little difference in performance for providing an asset as long as the ID of the asset is known. Since (i) requires no actual query processing beyond a simple filter, the difference in performance shows only the difference of providing data from an already opened file and data from a file which still has to be opened. In micro-benchmark (ii), the overhead of parsing thousands of text files is clearly noticable. Not only did our hardware experience the mentioned thermal stress, the execution took 2.2 hours. For this reason, we run this parser micro-benchmark only once. In contrast, *SciFi* was able to return the results in 84 ms with no run taking öonger than 85 ms. This difference is not surprising because *SciFi* does not require to fetch and scan GBs of data from disc, but only runs a query on the much smaller *MetaStore*. In micro-benchmark (iii), we used the further reduced data set for the parser. This way, the query executed within 14 minutes, which is still magnitudes longer than the time required to execute the query in *SciFi*. We conducted multiple different queries with a varying number of filter attributes and remained at a relatively constant execution time of 14 min when using the parser. The bandwidth of the disc access is the main bottleneck in this conventional approach and cannot be masked by a higher number of comparisons during parsing.

5. Related Work

Most asset stores for scientific data are domain-specific and limited to few document formats. Examples for domain-specific asset stores are XNAT [11] for neuroimaging, BIDS-manager for electrophysiology and neuroimaging [6], and OME/OMERO for bioimaging [12]. Some traditional database management systems (DBMS) offer extensions for selected domains, e.g. PostGIS is a PostgreSQL extension for geospatial data. However Bartoszewski et al. show that PostGIS is not in all cases superior to document databases when running equivalent queries [13]. There are also domain-independent DBMS which have been developed because of a need for such systems in data science. For instance, genetics researchers played a role in the initial idea of developing DuckDB [14], which is a relational system. Another kind of database systems are array databases. SciDB [15] is an array database which was developed with scientific use-cases in mind. Thus, DuckDB and SciDB both rely on a single storage layout and query method. Combining assets and metadata with one of these DBMS requires additional effort by the user which *SciFi* avoids. However, these systems are candidates for the backends of our approach as we have already demonstrated with DuckDB.

To the best of our knowledge, the only previously existing general purpose asset store for scientific data is DERIVA [7]. DERIVA is built as a collection of web services. In contrast to this, *SciFi* targets data management which happens locally and close to the user, i.e. on individual laptops and lab computers.

Further, there is a clear distinction between repositories and other data sources with added functionality for specific domains, e.g. the Protein Data Bank[5] that we used in our evaluation, and the underlying systems which handle such data. *SciFi* is a system solution, not a repository.

6. Outlook

SciFi has potential for different further developments. On the one hand, the usability can be further improved over the current early development stage. For instance, a graphical user interface, and the provision of binaries specialized for popular datasets, e.g. PDB or AlphaFold, are planned.

On the other hand, our asset store is a foundation for the development of an integrated data analysis (IDA) pipeline. Since such pipelines are usually bound by the ability to provide and analyze data fast and in a simple way, it is a logical step to apply a system using integrated databases which are optimized for exactly these tasks. The development towards an IDA pipeline can incorporate a number of steps and directions. One of these directions is the implementation of DataStore backends which are specialized on heavily used document types, e.g. hdf5. Another step could be the translation of the contents in the DataStore from a generic key-value pair into more structured content. For instance, there is already existing work about table recognition in spread sheets which treats also non trivial cases [16, 17, 18, 19]. The same approaches can be used for a more sophisticated automatic import of metadata. The provision of interfaces for other popular AI frameworks, e.g. Tensorflow or Keras, is another step towards a fully working IDA pipeline. Alternatively, the extension of the framework by commonly used algorithms, e.g. genetic algorithms, is a promising goal. Finally, interoperability with more complex and not necessarily local IDA pipelines, such as DAPHNE [20], can increase the use of *SciFi* in everyday lab life.

7. Conclusions

We presented *SciFi*, a framework for an embedded asset store for scientific data along with a sample application and an integration into PyTorch. This asset store closes a gap between existing solutions which are either domain-specific or dependent on a client-server architecture requiring administration and privileges for installation. In contrast to this, *SciFi* enables researchers to work not only faster because they have their data at hand, but also offline and on every available machine, mostly regardless of any limitations set by the IT department. Additionally, a combination of two databases with all their benefits is used to organize data without the requirement to learn a new query language. A connection between metadata and data is done automatically, such that no additional scripting is necessary for this task. We are confident that *SciFi* is a solid foundation for an embedded integrated data analysis (IDA) pipeline for use in the lab and at home office.

References

[1] P. Mooney, 2021 & 2022 kaggle data science & machine learning survey, 2021 - 2022.

[2] M. Raasveldt, H. Mühleisen, Data management for data science-towards embedded analytics., in: CIDR, 2020.

[3] M. D. W. et al., The fair guiding principles for scientific data management and stewardship, Scientific data (2016).

[4] W. Dempsey, I. Foster, S. Fraser, C. Kesselman, Sharing begins at home, arXiv preprint arXiv:2201.06564 (2022).

[5] H. M. Berman, J. Westbrook, Z. Feng, Gilliland, et al., The protein data bank, Nucleic acids research (2000).

[6] N. Roehri, V. S. Medina, A. Jegou, B. Colombet, B. Giusiano, P. Aurélie, F. Bartolomei, C.-G. Bénar, Transfer, collection and organisation of electrophysiological and imaging data for multicentre studies, Neuroinformatics (2021).

[7] R. E. Schuler, C. Kesselman, K. Czajkowski, Accelerating data-driven discovery with scientific asset management, in: e-Science, 2016.

[8] M. Raasveldt, Monetdblite: An embedded analytical database, in: SIGMOD, 2018.

[9] F. Griese, J. Kummer, P. L. Connor, M. Brüggen, L. Rustige, First radio galaxy data set containing curated labels of classes fri, frii, compact and bent, Data in Brief (2023).

[10] J. Jumper, R. Evans, A. Pritzel, T. Green, et al., Highly accurate protein structure prediction with alphafold, Nature (2021).

[11] D. S. M. et al., The extensible neuroimaging archive toolkit, Neuroinformatics (2007).

[12] J. R. Swedlow, I. G. Goldberg, K. W. Eliceiri, O. consortium, Bioimage informatics for experimental biology, Annual review of biophysics (2009).

[13] D. B. et al., The comparison of processing efficiency of spatial data for postgis and mongodb databases, in: Beyond Databases, Architectures and Structures, 2019.

[14] M. Raasveldt, H. Mühleisen, Talk: Duckdb - an embeddable analytical rdbms, 2020. URL: https://db.in.tum.de/teaching/ss19/moderndbs/duckdb-tum.pdf.

[15] M. Stonebraker, P. Brown, D. Zhang, J. Becla, Scidb: A database management system for applications with complex analytics, Computing in Science & Engineering (2013).

[16] E. K. et al., Table recognition in spreadsheets via a graph representation, in: DAS, 2018.

[17] Z. Chen, M. Cafarella, Automatic web spreadsheet data extraction, in: Proceedings of the 3rd International Workshop on Semantic Search over the Web, 2013.

[18] M. D. Adelfio, H. Samet, Schema extraction for tabular data on the web, VLDB (2013).

[19] J. E. et al., Deexcelerator: a framework for extracting relational data from partially structured documents, in: CIKM, 2013.

[20] P. Damme, M. Boehm, M. Dokter, K. Innerebner, R. Kern, Daphne: An open and extensible system infrastructure for integrated data analysis pipelines (2022).