SciFi - First Steps
User Documentation

August 2022

Contents

1 Welcome to SciFi

SciFi is an Asset Store which was designed to provide a high degree of freedom for the definition of assets and metadata. It consists of a C++ framework, an extensible shell, some sample applications, and a PyTorch extension with an according example. This documentation shows the use of two example applications: 1) An application instantiating a simple shell, and 2) The PyTorch example. We suggest to use these examples to get familiar with concept of SciFi and then extend the framework according to your needs.

We demonstrate our examples using an Ubuntu 20.04 LTS. Please make sure that the example binary (*simple*) and the python wrapper (*pythonwrapper/ scifi.cpython-38– <architecture> –<toolchain>.so*) are built. Refer to the readme for instructions on the build process. We will use the test dataset

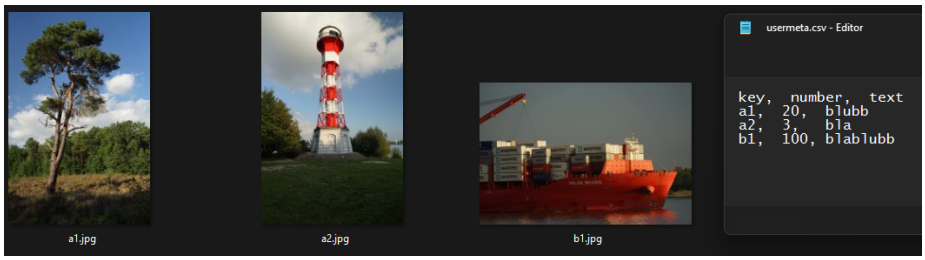

Figure 1: Our example dataset consists of 3 images and some metadata in a csv file

Figure 2: Welcome screen of the SciFi shell

hamburgimages provided in the folder *example_datasets*. The dataset contains a folder called *assets* with three images. The user-defined metadata can be found in the file *usermeta.csv* (see Figure 1).

If you want to get to the PyTorch example as fast as possible, go only through the sections 2, 2.1, 2.3, and 3.

2 The SciFi shell

The example application, which is built using the script *downloadAndBuild.sh*, is a simple shell. After a successful build, the binary *simple* can be found in the root folder of SciFi. Run this binary by calling `./simple` from a command line. You will be greeted like shown in Figure 2. A complete list of all available instructions can be printed by typing `h`. The shell closes by typing `q`.

If running the application for the first time, there will be no assets or metadata available. However, an empty MetaStore and an empty DataStore are created. The DataStore is a folder called *My_Assetstore*. It contains one or more

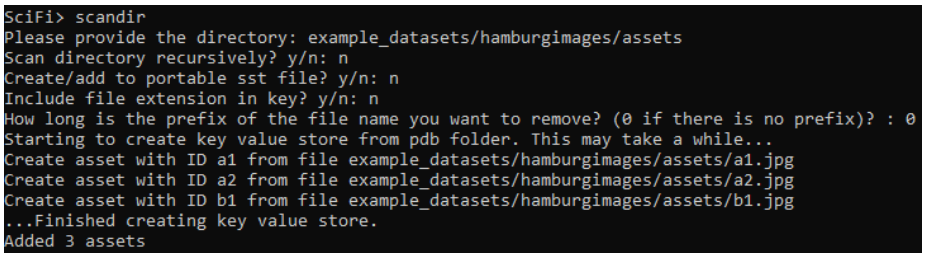

Figure 3: Ingesting assets into the DataStore

log files and configuration files. The MetaStore is a file called *Test_Metastore*. This is a sql file in the same format as used by SQLite. A second file called *Test_Metastore.wal* is a log file. To make some example data available, we will scan a directory for assets and read some metadata from a csv file in the following sections.

2.1 Ingesting assets into the DataStore

All assets are stored in the DataStore. To scan a directory for assets and ingest them into the DataStore, call *scandir*. The shell will ask for some necessary information as shown in Figure 3. In the following, we will go through all questions which have to be answered.

1. **Please provide the directory:** provide the directory you want to scan. This works with relative paths as well as with absolute paths. If you did not change the folder structure and are running the binary from the root folder, the example assets are found in `example_datasets/hamburgimages/assets`.

2. **Scan directory recursively? y/n:** You can either scan only the assets which are directly in the provided folder, or also scan all subfolders. There are no subfolders in the example. Type `n` to scan wthout including subfolders.

3. **Create/add to portable sst file? y/n:** There are portable and local DataStores. A local DataStore is used for working with assets. However, it would require copying the whole folder containing the DataStore to share the assets. In contrast to this, a portable DataStore consists only of a single file which can be coped and which includes all necessary information. A portable sst file can be imported into a local DataStore. We are going to work with the data instead of sharing it. So we ingest the assets directly into the local DataStore. Type `n` to do this.

4. **Include file extension in key? y/n:** Keys have to be unique. By default, a key is created from the filename without the file extension. However, keys have to be unique. Thus, it might be useful to include the

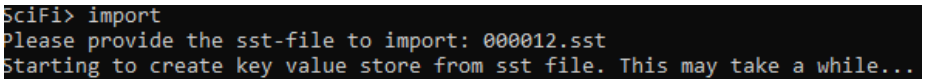

Figure 4: Import a portable DataStore file

file extension in your keys. This is also useful if the user-defined metadata uses the whole file name including the file extension.

5. **How long is the prefix of the file name you want to remove? (0 if there is no prefix)?** Some datasets we used for testing had filenames with prefixes, e.g. for indicating the format or origin. You may remove these prefixes by providing their length.

6. **Only for recursive scans: How many directories do you want to keep as part of the key? (0-9, type 0 if you are not sure, only works if you have this many subdirectories):** One of our test datasets included the name of the subdirectory in the key of the user-defined metadata. To accomodate for such cases, you are able to include the name of the subdirectories up to the depth defined here.

After all questions have been answered, your given directory is scanned and all found files are added as assets. Remember that key have to be unique. If a key already exists while scanning a new asset, you will be prompted with an error message. However, this does not stop the process of scanning the directory. If duplicates should be removed intentionally, you do not have to do anything. If this was an accident, the scan should be rerun with different properties for the key, e.g. including the name of the directory or not removing the prefix. For our example, this is not the case as there are no duplicates.

Once the scan is complete, the number of added assets is shown. If this is what was expected, the original files are not necessary anymore. Ideally, they are backed up and never looked at again. All data is stored in the directory of the DataStore, or in a single file if you chose to create a portable DataStore. The number of files in the directory of the DataStore can look overwhelming at first sight, but there is no reason to worry. There will never be more than 700 files, regardless of the number of assets. While adding new assets, files are automatically created and merged to move from a write optimized storage for adding assets to a read optimized storage for accessing assets.

2.2 Import a portable DataStore

Congratulations! Someone wants to share their data with you. Use `import` and provide the name of the file to ingest a portable DataStore file into your own Asset Store as shown in Figure 4. Depending on the size of the store and the number of assets, this might take a while. Once the import is done, you can use all assets the way you are used to. Already existing assets are not affected by the import.

```
SciFi> importmeta
Please provide the filename: example_datasets/hamburgimages/usermeta.csv
which column contains the key (provide the name): key
```

Figure 5: Reading metadata from a csv file

```
DROP TABLE IF EXISTS metadata
Success
BOOLEAN
[ Rows: 0]


CREATE TABLE IF NOT EXISTS metadata AS SELECT * FROM 'example_datasets/hamburgimages/usermeta.csv';
Count
BIGINT
[ Rows: 1]
3
```

Figure 6: SQL queries performed for reading metadata and the according output message.

2.3 Add user-defined metadata

A DataStore alone reduces the used disc space and can speed up the access to your assets. A MetaStore adds more functionality to your Asset Store. The easiest way to add metadata is to read it from a csv file. We will use the file *usermeta.csv* which includes some simple metadata. Note that the example shell expects the headers of the columns in the first line. Every line after the header is interpreted as a record. A more advanced interface for reading metadata is provided by the framework, but not used by the example shell.

The example file includes 3 attributes: *key*, *number*, and *text*, where we define *key* to be the unique key in the Asset Store. Note that the unique key has to match the keys of the assets. This is why there are different ways to customize the keys while importing new assets.

To import metadata from a csv file, type `importmeta`. You will then be asked for the filename and the name of the column which contains the unique key (see Figure 5. Then, a potentially already existing table with metadata is deleted and a new one created from the provided file. Data types are resolved whereever possible. The shell prompts the according SQL calls and the number of processed records as shown in Figure 6. In our case, 3 records were processed, one per key. After importing the metadata, SciFi will not require the csv file anymore but work with the MetaStore instead.

```
SciFi> ext
Please provide the file extension for assets (currently '.dat'): .png
Please provide the file extension for meta data (currently '.txt'): .txt
```

Figure 7: Changing the extension for returned assets and metadata

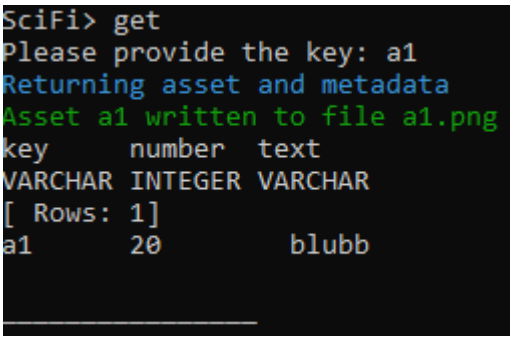

Figure 8: Return an asset by its key

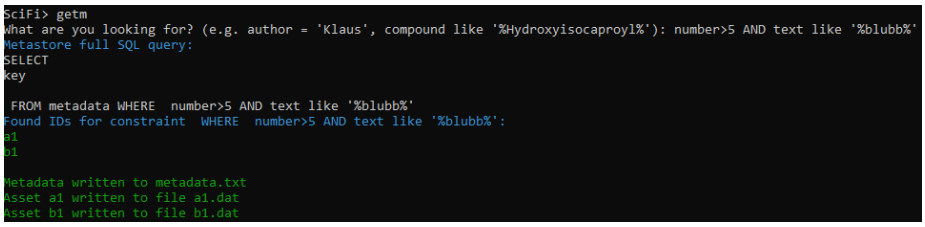

Figure 9: Return assets filtered by their metadata

2.4 Return assets

The example shell writes all assets to disc, into a folder called *results*. The framework also offers writing them to a temporary file system in main memory or to return them directly. However, the example shell does not use these features. Assets can be filtered by their key or by their according metadata. We will explain both cases.

2.4.1 Choose file extension for assets

Although the extension of a file is stored in the automatically generated metadata, it is currently not used to define the file extension of returned assets. Changing this is on our todo list, but for now the user has to pick a file extension. The standard extension for assets is .dat and for the corresponding user-defined metadata it is .txt. To change these defaults, call `ext` and provide your chosen extensions as shown in Figure 7.

2.4.2 Return assets by key

To return an asset by its key, call `get` and provide the key you are looking for. If an asset with this key is found, it will be written as a file into the results folder. The name of the file is derived from the key. The complete metadata for this key is also printed (see Figure 8.

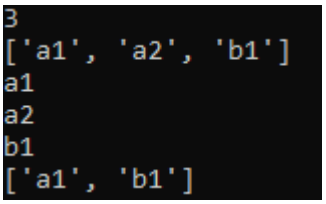

Figure 10: The output of our pytorch example.

2.4.3 Return assets by metadata

A useful feature of all Asset Stores is the filter for metadata. To find all assets which correspond to a certain definition of metadata, type `getm`. You can then provide the constraints for the metadata. In our example, there are 3 attributes which we can filter for: key, number, and text. It is possible to filter for multiple attributes by combining them via `AND` and `OR`. To filter for all assets where the number is larger than 5 and the text contains the sequence 'blubb', use `number>5 AND text like '%blubb%'`. The executed SQL query will be shown followed by all keys which match the condition(s). All according assets will be written as files to the results folder. The metadata of these assets is written into a text file called *metadata.txt*. If the condition 'all' is provided, all assets are returned.

3 The *SciFiDataset* for PyTorch

It is possible to directly access data stored in SciFi without the need to write assets to discs. For this purpose, there is an extension of the *dataset* class in PyTorch. The derived class is called *SciFiDataset* and can be found in the file *SciFiDataset* in the folder *pythonwrapper*.

Additionally to the functions *__getitem__* and *__len__* which are present in all PyTorch dataset extensions, there are the functions *getIDsByIndex(min,max)* and *getIDsByMeta(meta)*. The script *DatasetExample.py* demonstrates their usage. A correct output for the example dataset is shown in Figure 10. In the following, the individual steps of the script are explained.

Note that the wrapper library has to be built before the *dataset* extension can be used. Please refer to the readme for instructions on the build process.

3.1 Instantiate a dataset

To instantiate the SciFiDataset, the names of the DataStore and the MetaStore have to be provided. If they are not in the same directory as the script you are running, the path also has be provided. This can either be a relative or an absolute path. In our example, we provide a relative path (line 4 in Listing 1).

```
1 #!/usr/bin/env python3
2 from SciFiDataset import SciFiDataset
3
4 scifi_dataset = SciFiDataset(datasetname="../
      My_Assetstore", metaname="../Test_Metastore")
```

Listing 1: Instantiate a dataset

3.2 Test access

Once the dataset is instantiated, all assets can be accessed in a map-style manner via their unique key. Line 1 in Listing 2 shows an example where the key is *a1*. To test if an asset is returned exactly the way it is stored in the asset store, e.g. that there are no conversion errors, we write the returned asset to a file called *test.jpg* (line 3 and 4).

```
1 sample=scifi_dataset["a1"]
2
3 with open('test.jpg', 'wb') as f:
4     f.write(sample)
```

Listing 2: Test the access to the assets

3.3 Iterate over assets

Although we created a map-style dataset, data should be iterable to some degree. To iterate over all assets, the number of available assets is required for which we use the *__len__* function. The length, i.e. the number of assets in the dataset, is used to return all keys by using the function *getIDsByIndex(min,max)* (line 2 in Listing 3). Since keys are not necessarily numeric, it is not possible to iterate the dataset using the length directly. Internally, *getIDsByIndex* queries all keys from the MetaStore, sorts them alphabetically, and selects the given (min,max) window from the result. If *min* is 0 and *max* is the number of assets, the window includes all results.

The returned list of keys can now be used to iterate the dataset as shown in the loop starting at line 5. For testing purposes, we also print the key in every iteration (line 7) and write the asset to a file (line 8 and 9).

```
1 print(len(scifi_dataset))
2 list = scifi_dataset.getIDsByIndex(0,len(scifi_dataset))
3 print (list)
4 j=0
5 for i in list:
6     sample=scifi_dataset[i]
7     print(i)
8     with open(str(j)+'.jpg', 'wb') as f:
```

```
9           f.write(sample)
10      j=j+1
```

Listing 3: Iterate over all assets

3.4 Filter by metadata

One of the most helpful features of Asset Stores is the ability to filter assets for certain constraints in their metadata. To achieve this, the function *getIDsByMeta* can be used. The constraint for the metadata is provided as a function parameter as shown in line 1 in Listing 4. The returned list can then be used to iterate all affected assets as already demonstrated in Section 3.3.

```
1 metalist = scifi_dataset.getIDsByMeta("number>10")
2 print(metalist)
```

Listing 4: Filter keys by metadata