B-Store Documentation

Release 1.1.1

Kyle M. Douglass

January 26, 2017

Contents

Author Kyle M. Douglass

Contact kyle.m.douglass@gmail.com

organization École Polytechnique Fédérale de Lausanne (EPFL)

revision $Revision: 3 $

date 2017-01-24

Lightweight database management and analysis tools for single molecule microscopy.

CHAPTER 1

Using B-Store

1.1 Quick Start

Author Kyle M. Douglass

Contact kyle.m.douglass@gmail.com

organization École Polytechnique Fédérale de Lausanne (EPFL)

revision Revision: 3

date 2016-11-06

abstract This quick start guide shows how to get up and running with B-Store as quickly as possible.

Table of Contents

1.1.1 Installation

Anaconda

Installation is most easily performed using the Anaconda package manager. Download Anaconda for Python 3 (or Miniconda) and run the following commands in the Anaconda shell:

```
conda update conda
conda config --append channels soft-matter
conda create -n bstore -c kmdouglass bstore
```

The above commands add a custom channel to the package manager (soft-matter) and give it lower priority over the default channel. Then, a new conda environment named bstore is created and the bstore package is installed from the kmdouglass channel.

If you would like to use Jupyter Notebooks–which aren't required–, then be sure to run these commands after installing B-Store:

```
conda install jupyter nb_conda
```

Every time you want to run bstore, ensure that you are working in the bstore environment with

```
activate bstore
```

on Windows and

```
source activate bstore
```

on Linux and Mac.

Installation from Source

Alternatively, the source code for B-Store may be cloned from https://github.com/kmdouglass/bstore/. A list of dependencies may be found inside the *requirements.txt* file inside the repository.

The master branch contains code that has been more thoroughly tested than any other branch. The development branch contains the version of the code with the latest features but is more likely to suffer from bugs.

1.1.2 Getting Started

Background

The B-Store workflow is divided between these two tasks:

1. Sort and place all the files from a single molecule localization microscopy (SMLM) acquisition into a single file known as a *Datastore*.
2. Automatically access this datastore for batch analyses.

B-Store uses popular scientific Python libraries for working with SMLM data. Most notably, it uses Pandas DataFrames for working with tabulated localization data and the standard json module for handling metadata. Images are treated as NumPy arrays whose image metadata can be read from tiff tags (OME-XML and Micro-Manager metadata are currently supported). Reading and writing from/to HDF files is performed with h5py (though Pandas uses PyTables for a few operations).

What all this means is that if you can't do something with B-Store, chances are you can implement a custom solution using another Python library.

Jupyter Notebook Examples

If you want to learn more after working through the quick-start guide, then you can find examples inside the Jupyter Notebooks at the B-Store GitHub repository.

Jupyter Notebooks are a great way to interactively work with B-Store when writing code and are very common in the scientific Python community. They are free, powerful, and provide a convenient way to document your work and share it with others. Alternatively, you may use any other Python interpreter to work with B-Store functions.

B-Store Test Datasets

The B-Store test files repository contains a number of datasets for B-Store's unit tests. These datasets may also be used to try out the code in the examples or in this guide.

Workflow Summary

B-Store is a collection of tools for working with SMLM data. You may interact with these tools in two different ways:

1. by using the GUI, and
2. by writing Python code

Once you have a set of HDF files, you may open them in any software package or language that supports HDF, such as MATLAB.

1.1.3 Build a HDF Datastore with the GUI

To start the GUI, navigate to the console window (or Anaconda prompt). If you installed B-Store from Anaconda, be sure to activate the bstore environment using whatever name you chose when creating it:

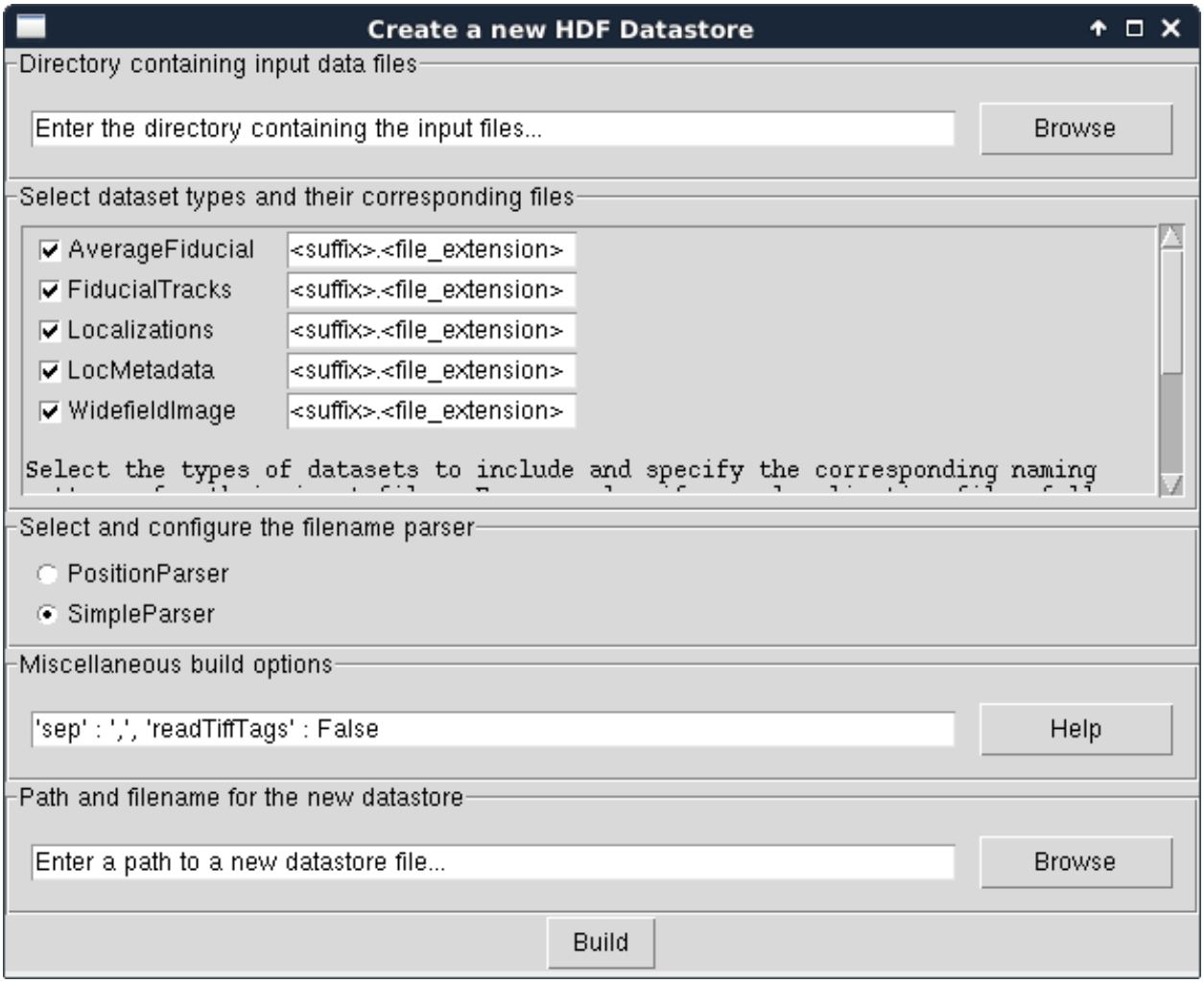
```
source activate bstore
```

If you're on Windows, don't use the word **source**.

Once activated, simply run the program by typing:

```
bstore
```

In the window that appears, select **File > New HDF Datastore...**. The following new dialog will appear:



First, choose the directory where the raw data files and subdirectories are located. We will use the test files for the SimpleParser for this example. Please note that this directory **and all of its subdirectories** will be searched for files ending in the suffix.filename_extension pattern specified in the next field.

Next, select what types of datasets should be included in the datastore. For this example, check **Localizations**, **LocMetadata**, and **WidefieldImage** and uncheck the rest. Set the filename extension of Localizations, LocMetadata, and WidefieldImage to **.csv**, **.txt**, and **.tif**, respectively. This will tell the build routine what files correspond to which types of datasets.

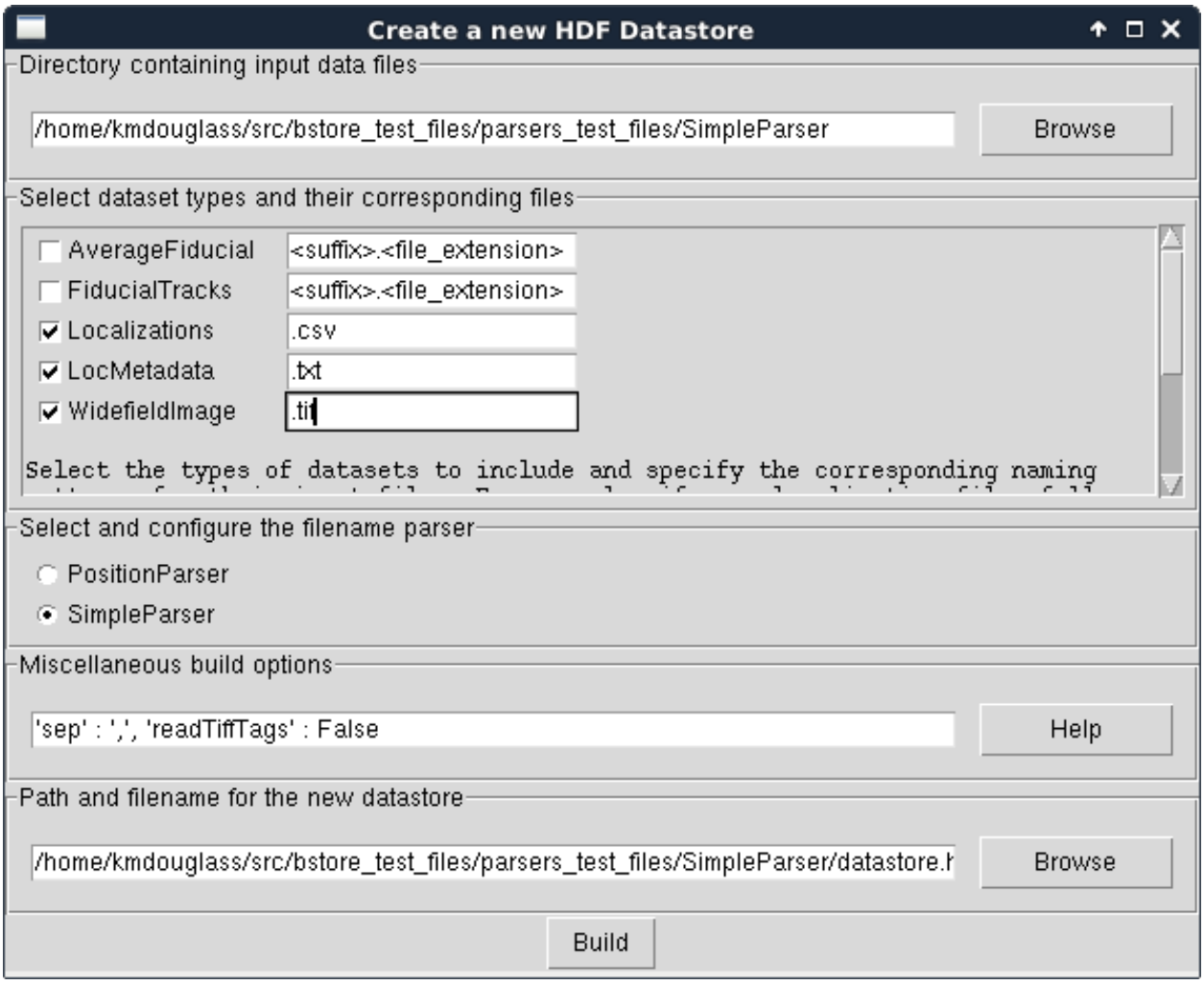
If your files have a special identifier in their filename, like **locs** for localizations, you can enter search patterns like **locs*.csv**. The asterik (*) will act as a wildcard such that files like *cells_locs_2.csv* or *Cos7_alexa647_locs.csv* would be found during the file search.

Leave the parser set to **SimpleParser**. A parser converts a filename into a set of DatasetIDs that will uniquely identify it inside the Datastore.

After this, leave the Misc. options as they are. This box allows you to manually specify options for reading the raw data files. 'sep' for example is the separator between columns in a .csv file. If you have a tab-separated file, change ',' to 't' (t is the tab character). Change 'readTiffTags' from False to True if you have Micro-Manager or OME-XML metadata in your tif image files. Please note that this may fail if the metadata does not match the filename like, for example, what would happen if someone renamed the file.

Finally, use the Browse dialog to select the name and location of the HDF datastore file in the top-most field.

The window should now look like this:



Click the **Build** button and when it completes, you should have a nice, new HDF Datastore with your data files structured safely inside it.

Misc. Build Options

The misc. build options, like *sep* and *readTiffTags*, are passed to each Dataset's method for reading data from files. They are specified in the same notation as Python dictionaries except they omit the curly braces. Each one is optional, so you need not specify any of them.

The name of each option must be surrounded in single quotation marks. The value for each option is a Python datatype and is separated from the option's name by colon. All option/value pairs are separated by commas. True and False are case-sensitive. Strings are also surrounded by single quotes.

The current list of options is:

1. **sep** - The column separator in the raw text csv files. Common values include commas ',' and tabs '\t'.
2. **readTiffTags** - Determines whether tif image metadata should be read and recorded in the HDF datastore. Accepts either *True* or *False*. Note that this may fail to read the tif images if the filename does not match the metadata.

For Pythonistas: The evaluation of the string inside this Entry is performed with ast.literal_eval(). It is a secure method, unlike *eval()*, but can only evaluate basic Python datatypes.

1.1.4 Programming with B-Store

B-Store also has an API which allows you to write scripts and Python code to integrate B-Store into your custom workflows.

Parsing Files to assign Dataset IDs

A B-Store *Datastore* is a storage container for things like sets of localizations, widefield images, and acquisition metadata. Each dataset in the datastore is given a unique ID. A parser reads your data from files and gives it a meaningful set of datastore IDs. For example, if you have localizations stored in a comma-separated text file named *HeLaL_Control_1.csv* and you use the built-in SimpleParser, then your dataset will have the following ID's:

1. *prefix* - 'HeLaL_Control'
2. *acqID* - 1

You can follow along by entering the following code into the Python interpreter of your choice and using the SimpleParser test files.:

```
>>> import bstore.parsers as parsers
>>> sp = parsers.SimpleParser()
>>> sp.parseFilename('HeLaL_Control_1.csv', 'Localizations')
>>> sp.dataset.datasetIDs
{'acqID': 1, 'prefix': 'HeLaL_Control_1'}
```

Here, *Localizations* refers to a specific dataset type used by B-Store to read and write localization data.

B-Store comes with two built-in parsers: SimpleParser and PositionParser. The SimpleParser can read files that follow the format **prefix_acqID.(filename extension)**. The very last item of the filename is separated from the rest by an underscore and is always assumed to be an integer. The first part of the filename is a descriptive name given to the dataset.

The PositionParser is slightly more complicated, but gives you greater flexibility over how your filenames are read. It assumes that each dataset ID is separated by the same character(s), such as _ or -. You then specify the integer position (starting from zero!) that each ID is found in.

For example, say you have a filename like **HeLa_Data_3_2016-05-12.csv**. You want **HeLa** to be the prefix, **Data** to be ignored (not used to assign an ID), **3** to be the acquisition ID number, and **2016-05-12** to be the date. These correspond to positions 0, 1, 2, and 3 in the filename, respectively, and the separator is an underscore (_). You would initialize the PositionParser like this:

```
>>> pp = parsers.PositionParser(positionIDs = {
>>>     0 : 'prefix', 2 : 'acqID', 3 : 'dateID'})
```

Changing the separator of 'positions' is also easy: simply specify a *sep* parameter to the PositionParser's constructor. We can change the seperator to hyphen underscore (-_) like this:

```
>>> pp = parsers.PositionParser(
>>>>    positionIDs = {
>>>         0 : 'prefix', 2 : 'acqID', 3 : 'dateID'},
>>>     sep = '-_')
```

If you require a customized parser to assign ID's, the Jupyter Notebook tutorial on writing custom parsers is a good place to look.

Building a Datastore

You will typically not need to work directly with a parser. Instead, the B-Store datastore will use a specified parser to automatically read your files, assign the proper ID's, and then insert the data into the database.

Let's say you have data from an experiment that can be parsed using the **SimpleParser**. (Test data for this example may be found at https://github.com/kmdouglass/bstore_test_files/tree/master/parsers_test_files/SimpleParser .) First, we setup the parser and choose the directory containing files and subdirectories of acquisition data.:

```
>>> from bstore import database, parsers
>>> from pathlib import Path
>>> dataDirectory = Path('bstore_test_files/parsers_test_files/SimpleParser')
>>> parser = parsers.SimpleParser()
```

Next, we create a name for the HDF file that a HDFDatastore points to. The HDFDatastore class will be used to interact with and create B-Store databases.:

```
>>> dsName = 'myFirstDatastore.h5'
```

After this, we tell B-Store what types of files it should know how to process:

```
>>> import bstore.config as cfg
>>> cfg.__Registered_DatasetTypes__ = [
        'Localizations', 'LocMetadata', 'WidefieldImage']
```

Localizations, **LocMetadata**, and **WidefieldImage** are built-in dataset types. Telling B-Store what types of files to look for helps prevent it from mistakenly thinking a random file that accidentally entered the directory tree contains SMLM data.

Finally, we create the database by sending the parser, the parent directory of the data files, and a dictionary telling the parser how to find localization files to the **build** method of myDS. Note that myDS must be created inside a *with...as...* block to ensure the file is properly opened and closed. The *put()* and *build()* methods of HDFDatastore both require the use of *with...as...* blocks; all other methods do not.:

```
>>> with database.HDFDatastore(dsName) as myDS:
>>>     myDS.build(sp, dataDirectory, {'Localizations'  : '.csv',
                                       'LocMetadata'    : '.txt',
                                       'WidefieldImage' : '.tif'})
6 files were successfully parsed.
```

This creates a file named myFirstDatabase.h5 that contains the 6 datasets contained in the raw data. (If you want to investigate the contents of the HDF file, we recommend the HDFView utility.)

To specify exactly how data is read from your raw files, please see Tutorial 4 in the examples. This will teach you how to user Readers to read data in custom file types into Python and subsequently place them inside the HDFDatastore.

Batch Analysis from a B-Store Database

Another great utility of the B-Store database is that it enables batch analyses of experiments containing a large number of acquisitions containing related but different files.

As an example, let's say you want to extract all the localization files inside the database we just created and filter out localizations with precisions that are greater than 15 nm and loglikelihoods that are greater than 250. We do this by first building an analysis pipeline containing **processors** to apply in sequence to the data.:

```
>>> from bstore import batch, processors
>>> uncertaintyFilter = processors.Filter('uncertainty', '<', 15)
>>> llhFilter = processors.Filter('loglikelihood', '<=', 250)
>>> pipeline = [uncertaintyFilter, llhFilter]
```

Next, use an **HDFBatchProcessor** to access the database, pull out all localization files, and apply the filters. The results are saved as .csv files for later use and analysis.:

```
>>> bp = batch.HDFBatchProcessor(dsName, pipeline)
>>> bp.go()
Output directory does not exist. Creating it...
Created folder /home/douglass/src/processed_data
```

Inside each of the resulting subfolders you will see a .csv file containing the filterd localization data. A more complete tutorial may be found at https://github.com/kmdouglass/bstore/blob/master/examples/Tutorial%202%20-%20Introduction%20to%20batch%20processing.ipynb .

1.1.5 Getting Help

If you have any questions, feel free to post them to the Google Groups discussion board: https://groups.google.com/forum/#!forum/b-store

Bug reports may made on the GitHub issue tracker: https://github.com/kmdouglass/bstore/issues

1.2 B-Store Datastores

Author Kyle M. Douglass

Contact kyle.m.douglass@gmail.com

organization École Polytechnique Fédérale de Lausanne (EPFL)

revision $Revision: 1 $

date 2017-01-24

abstract The logic behind B-Store datastores is presented in this document. The HDF file type is briefly explained, followed by the organization of data within the database.

Table of Contents

1.2.1 Introduction to B-Store Datastores

A single high-throughput SMLM experiment can generate hundreds or even thousands of different files containing different types of data. Analyzing this data requires that the files are sorted and organized in a well-structured way

that is understandable by both humans and machines. A B-Store datastore fulfills this role as a structured container for heterogeneous SMLM data.

In basic terms, a B-Store *datastore* is a collection of individual *datasets*. Each dataset possesses identifiers that uniquely identify it within the datastore. A dataset also provides a container for the actual experimental data that it is holding, such as localizations or widefield images.

1.2.2 Datasets

A Dataset is a single, generalized dataset that can be stored in a B-Store datastore. It is "general" in the sense that it can represent one of a few different types of data (e.g. localizations, metadata, or widefield images). A specific type of dataset is called a *DatasetType*. A DatasetType knows what readers it may use to read raw input files from the disk and how to get and put data of its own type from and into a datastore. Unlike the Datastore, which sorts and organizes Datasets, the DatasetType encapsulates all the knowledge about data input and output.

There are currently five dataset types (examples of their raw input are in paranetheses):

1. Localizations (tabulated localization data in raw text, csv format)
2. LocMetadata (information about how the localizations were generated in JSON format)
3. WidefieldImage (gray scale images; contains OME-XML metadata and Micro-Manager metadata)
4. FiducialTracks (localizations belonging to individual fiducials in csv format)
5. AverageFiducial (the average drift trajectory from many fiducials)

If you require a raw input type or a general DatasetType that is not listed here, B-Store can be easily extended to support it. Please let us know on the forum.

Dataset IDs

A dataset is uniquely defined by the following fields (the first four–prefix, acqID, datasetType, attributeOf–are required).

prefix A descriptive name given to the dataset.

acqID An integer that specifies the acquisition number of the dataset.

datasetType A string. Must be one of the types listed above. The a type must be in the list __Registered_DatasetTypes__ in config.py to be used.

attributeOf A string. Must be one of the dataset types listed above. This is the name of a type of dataset that this one describes.

channelID (optional) A string that specifies the fluorescence channel that the dataset was acquired in.

dateID (optional) A string in the format YYYY-MM-DD. This is for identifying the same field of view taken on different days.

posID (optional) A one or two-element tuple of integers specifying the position of the field of view of the dataset.

sliceID (optional) An integer identifying the the axial slice of the dataset.

replicateID (optional) An integer identifying a replicate or biological repeat. This is used when a dataset has the same IDs as another but comes from an independent sample.

Hierarchy of Dataset IDs

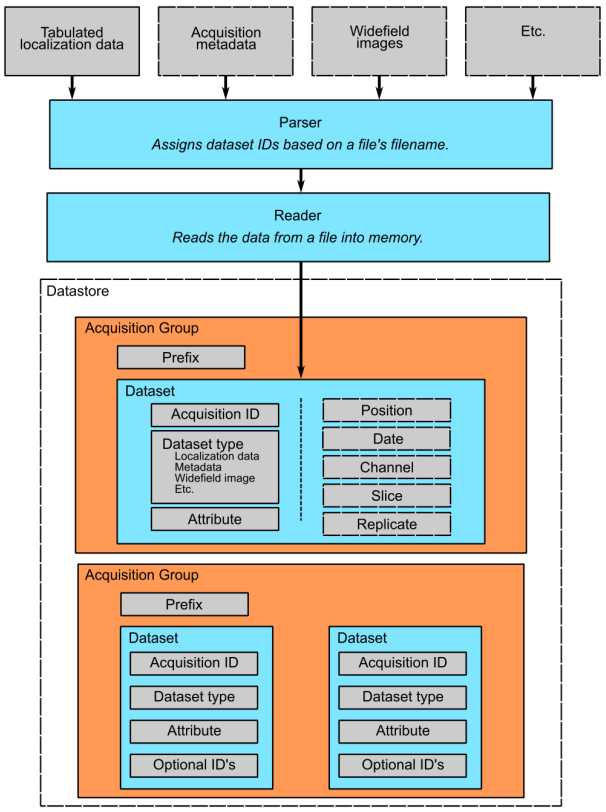
All datasets with the same prefix are organized into the same **acquisition group**. Within an acquisition group, datasets are specified according to their acqID.

For example, let's say we take three widefield images of Cos7 cells from the same coverslip during the same experiment. In the datastore, each image will have the same prefix, such as 'Cos7'. The individual images however will have three different acqID's. (Most likely they will be 1, 2, and 3, but they need not start at 1 or be sequential.)

If two datasets have the same prefix and acqID but different datasetType's, then they will be understood to have come from the same field of view. This allows widefield images to be grouped with their corresponding localizations within the database. As an example, we might have two datasets in our datastore where both have 'HeLa' as a prefix and 1 as the acqID, but one has 'Localizations' as its datasetType and the other 'WidefieldImage'.

Finally, the optional identifiers can further divide datasets that have the same prefix, acqID, and datasetTypes.

A diagram that explains this hierarchy is seen below. On top, you have your raw data files as inputs to a parser, which both assigns dataset IDs based on the files' filename. A Reader converts the data into a format suitable for insertion into the database. A single acquisition group is identified by a **prefix**. Within this group, each dataset has a unique **acqID** and **datasetType** to set it apart from other datasets within the same group. Finally, the other optional IDs give you more control over how the data is organized within the group.



The Role of Parsers in Datastores

As mentioned above, a B-Store parser is an object that assigns dataset IDs to a dataset based on the filename of the file containing the data.

Since different labs often have very different ways to generate their data, parsers were designed to be very flexible objects. The only requirement of a parser is that it implements the functions described by the Parser metaclass; these

functions specify the kinds of outputs a Parser must provide. The types of inputs, however, are not specified. This means that you can write a parser to convert any type of data that you would like into a dataset (as long as it fits within one of the datasetTypes). Furthermore, exactly how dataset IDs are assigned remains up to you. If you want your parser to label every single dataset with a prefix of 'Bob' then you can do that, though obviously the utility of such a feature will be in question.

This flexibility comes at a cost, however. If the built-in parsers do not work for your data, then it will be necessary to write your own. An example of how to do this is provided as a Jupyter notebook example.

The Role of Readers in Datastores

Readers do the actual work of reading the data inside a file into memory. When building a Datastore, a different reader may be specified for each dataset to allow B-Store to read data from a large range of file formats. Generic readers like CSVReader and JSONReader are provided for reading from generic file formats.

1.2.3 HDF Datastores

The HDFDatastore class allows for the creation of a datastore inside a HDF container. HDF is a high-performance file type used in scientific and numerical computing. It is considered a standard file type in scientific circles and is widely supported by many programming environments. One advantage of HDF containers is that you are not required to use B-Store code to access the data in a B-Store datastore. Any software that can read or modify HDF files will do.

HDFDatastore objects support many features of Python sets, like list comprehensions, filtering, and iteration.

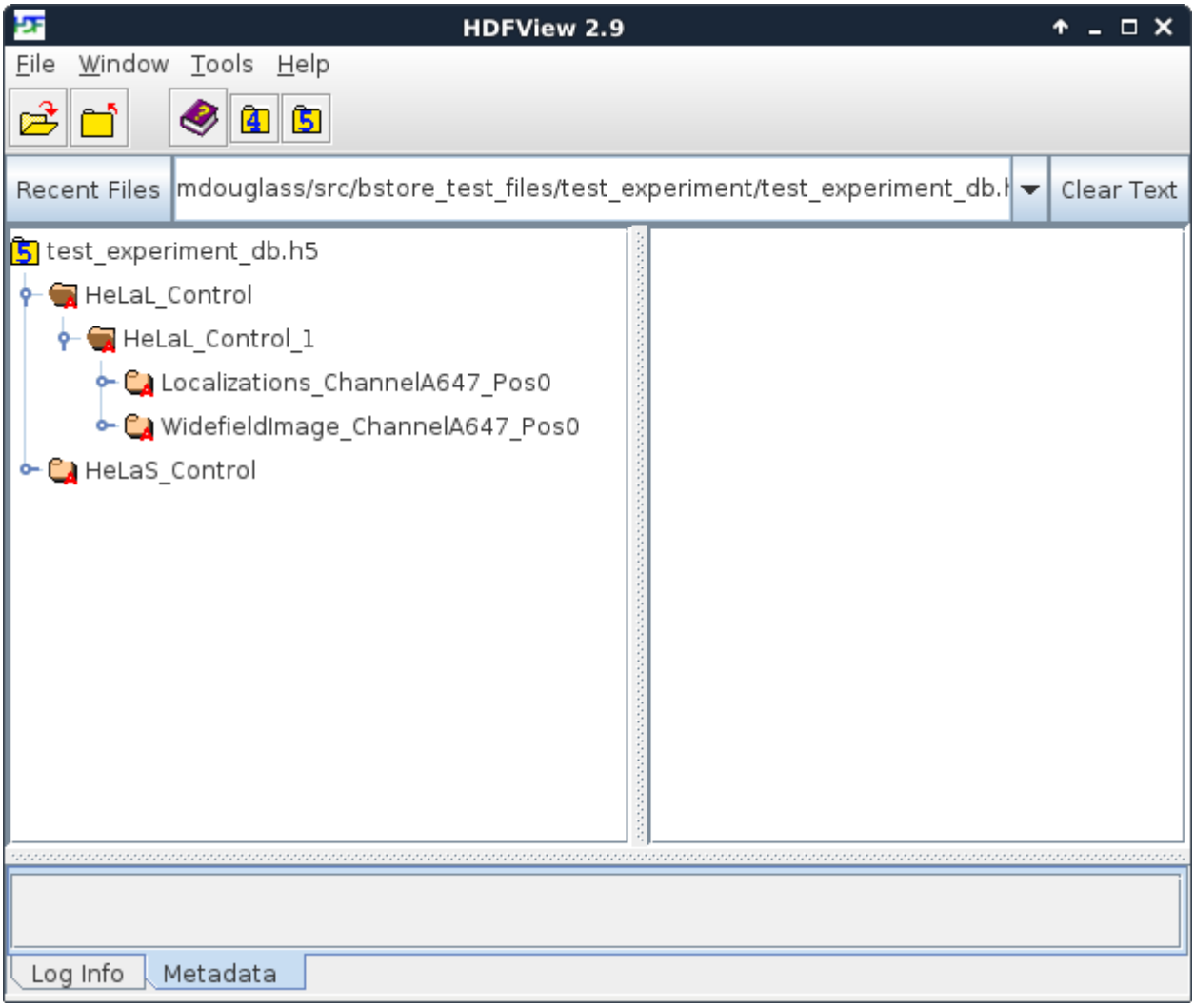
HDFView

HDFView is a useful utility for viewing the contents of a HDF container. It is freely available and recommended for trouble shooting.

We will use screenshots taken from HDFView to explain how data is sorted inside a B-Store datastore.

Organization within an HDF datastore

The figure below is a screenshot from HDFView of the B-Store test database located in test_experiment/test_experiment_db.h5 in the B-Store test files repository. On left side of the window, you can see a hierarchy of the groups stored inside this database. There are two acqusition groups with prefixes **HeLaL_Control** and **HeLaS_Control**. Inside the HeLaL_Control group, you can see that there is one single acquisition (labeled with an **acqID** of 1).



This group contains three different datasets: localizations (Localizations_ChannelA647_Pos0), a widefield image (WidefieldImage_A647_Pos0), and metadata describing how the localizations were obtained. (The metadata is not directly visible in this image because it's stored as attributes of the Localizations_ChannelA647_Pos0 group.) Each dataset has two optional identifiers: a **channelID** of A647 and a **posID** of 0. The dataset keys–if they are specified–follow the format **datasetType_channelID_posID_sliceID_dateID_replicateID**. Because no sliceID, dateID, or replicateID is specified, they are absent from the name of the group.

Date ID's are specified as strings in the format 'YYYY-MM-DD'.

Position ID's support single integer ID's as one-tuples *(0,)* and two integer ID's as two-tuples *(1,4)*.

Example of a full HDFDatastore key

A HDFDatastore key using all the ID's possible looks like:

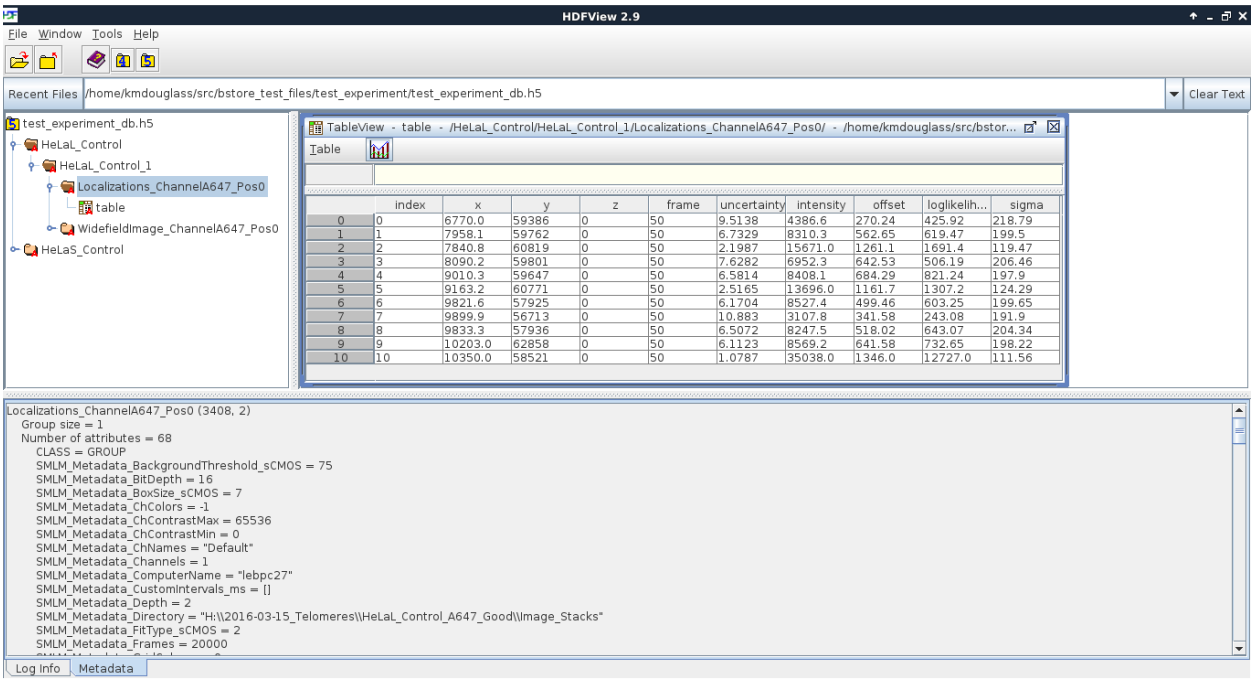
```
HeLa_Control/HeLa_Control_76/Localizations_ChannelA750_Pos1_Slice5_Date20161211_Replicate5
```

The dataset IDs matching this key are **prefix**: HeLaControl, **acqID**: 76, **datasetType**: Localizations, **channelID**: A750, **posID**: 1, **sliceID**: 5, **dateID**: 20161211, **replicateID**: 5.

If posID was specified with two integers, such as *(1,4)*, it the corresponding part of the key would look like *Pos_001_004*.

The dateID only has hyphens between the year, month, and day in Python; they are removed when writing to the HDF datastore.

As seen in the next figure, the actual localization data is stored as a table inside the Localizations_ChannelA647_Pos0 group. Metadata is attached as HDF attributes of the group; their values are in JSON format. Attributes have the same key as the dataset they belong to; if this dataset does not exist in the HDF file, neither will the metadata attributes. All attributes start with the string defined in the variable __HDF_Metadata_Prefix__ in config.py.



This mode of organization was chosen for a few reasons:

1. The data is organized in a way that is easily read by both humans and machines. This means we can understand the organization of the data without any knowledge of how the datastore was created.
2. B-Store dataset IDs can be inferred from the HDF key that points to the data. Machines can parse the HDF key to extract the dataset IDs, which is done, for example, when the function HDFDatastore.query() is executed.
3. We take advantage of features provided by the HDF format, such as attributes and groups.

1.3 Analysis Routines in B-Store

Author Kyle M. Douglass

Contact kyle.m.douglass@gmail.com

organization École Polytechnique Fédérale de Lausanne (EPFL)

status Revision: 0

date 2016-08-06

abstract A brief overview of performing simple and batch analyses in B-Store is provided.

Table of Contents

1.3.1 Analyzing SMLM Experiments with B-Store

First and foremost, B-Store is a tool for structuring data from SMLM experiments. With structured data, analysis of large datasets becomes easier because we can write programs to automatically take just the data we want and process it or make reports. The data is always organized in the same way, so our analysis routines can be easily adapted when new data arrives.

B-Store provides analysis routines as a secondary feature. Many software packages exist for analyzing SMLM data, and B-Store is not intended to replace them. Rather, B-Store provides common processing routines as a convenience–such as filtering or merging localizations–and less common processing routines for specialized analyses performed in the authors' laboratories.

Batch Processing

B-Store currently provides two batch processors for working with SMLM data: HDFBatchProcessor, for extracting data from B-Store HDF Datastores and processing them, and CSVBatchProcessor, for applying the same processing pipeline to .csv files spread across a directory tree.

The operation of a batch processor is simple: first, it accepts an analysis pipeline and a datastore or directory that contain at least one file corresponding to an SMLM dataset. The pipeline is a list of B-Store processors that modify a DataFrame containing localizations. Each processor is applied to a single dataset sequentially, starting from the first processor in the list.

Next, the batch processor accumulates a list of all the localization files in the database. If using the CSVBatchProcessor, it finds all files ending in the string parameter *suffix*. For example, if your localization files end in *locResults.csv*, you can set *suffix = 'locResults.csv'* and the batch processor will find these files in the specified folder **and all subfolders.** If using the HDFBatchProcessor, you can specify localization files using the *searchString* parameter.

Once the list of datasets is built, the batch processor loops over each dataset, applying the processors in the pipeline one at a time to the DataFrame. Currently, the output results are written to new .csv files in a folder specified in the *outputDirectory* parameter to the constructor of both batch processors. This feature allows you to perform analyses with different pipelines on the same database.

For an example of how to perform batch processing in B-Store, see the Jupyter notebook tutorial.

Analyzing Single Datasets

Single datasets may be retrieved from a B-Store database for analysis using the get() method of the Database class.

1.3.2 Processing Localizations and other Data

Processors

A processor is a simple class for processing localization datasets. Its behavior is controlled by zero or more attributes that are set in the processor's constructor. A processor is callable in that it is used like a function; when doing so, it **always accepts a single Pandas DataFrame as an input**.:

```
>>> import bstore.processors as proc
>>> myFilter = proc.Filter('precision', '<', 15)
>>> filterData = myFilter(df)
```

In the above example, we create a filter processor called *myFilter* whose constructor takes three arguments: the name of column to filter on, a string specifying the comparison operator (in this case less-than) and a numeric value. All rows in the 'precision' column will have values less than 15 after this filter is applied.

After creating the processor, you apply it to a Pandas DataFrame by using it like a function. In the above example, we pass a DataFrame named *df* to *myFilter* and store the processed DataFrame in *filterData*.

When creating your own processor, you can achieve this function-like behavior of a class by specifying the behavior inside the class's *__call__()* method. For more information, see the Python documentation.

A complete list of processors and their behavior may be found in the processor module index.

Multi-Processors

Multi-processors are similar to processors, except for two points:

1. they accept multiple inputs instead of a single DataFrame, and
2. they may take user-input and thus may not necessarily be used in batch processing.

Two examples of multi-processors are AlignToWidefield and OverlayClusters. *AlignToWidefield* determines the global offset between localizations and a widefield image by using a simple FFT-based cross-correlation routine.

OverlayClusters is a very useful analysis tool for displaying clustered localizations on top of a widefield image. This tool may be used to navigate through different clusters of localizations, manually filter clusters from the dataset, and to append numeric labels to clusters for manual segmentation. Generally, *AlignToWidefield* is before *OverlayClusters*.

1.3.3 Performing Analyses Outside of B-Store

B-Store databases use the HDF format and are therefore readable by many scientific libraries. You may analyze your data in any of these if the B-Store analysis tools do not suit your purposes.

1.4 File I/O

Author Kyle M. Douglass

Contact kyle.m.douglass@gmail.com

organization École Polytechnique Fédérale de Lausanne (EPFL)

revision $Revision: 0 $

date 2017-01-24

abstract A brief explanation of file input and output in B-Store.

Table of Contents

1.4.1 I/O in B-Store

Overview

B-Store uses two types of objects to read files from the disk that contain localization microscopy data:

1. Parsers
2. Readers

A Parser reads the filename of a file and assigns IDs to it that are used to identify the dataset inside a HDF file. A Reader reads the actual data contained in the file and converts it to an internal Python datatype. This datatype serves as an intermediary step before saving the data to the HDF file.

Readers are convenience tools that allow the same functionality for reading files to be applied to multiple datasetTypes. For instance, the *Localizations*, *FiducialTracks*, and *AverageFiducial* datasetTypes are all internally represented as Pandas DataFrames. Readers also allow B-Store to be easily extended to new types of files. The purpose in this is that users can store their localization microscopy data regardless of the software program that generated it. All that would be needed would be a new Reader instance that would know how to interpret the data in the files.

Important Note

Reader functionality was added in version 1.1.0 and is not yet integrated with all dataset types. This will change in future versions.

There is also no arbitrary file output from the HDF files as of version 1.1.0, except for that generated by CSVBatchProcessor. This too should change in upcoming versions.

Built-in Parsers

Two types of Parsers are currently built-in to B-Store:

1. SimpleParser
2. PositionParser

The SimpleParser interprets files names in the format PREFIX_ACQID.<file_type> where the prefix and acquisition ID are separated by an underscore. The PositionParser splits up a filename by a specified character, e.g. '_', and assigns DatasetIDs based on the elements of the filename occupying integer positions separated by this character, starting with 0 at the left-most position.

Built-in Readers

There are currently two Readers built-in to B-Store:

1. CSVReader
2. JSONReader

These readers use functionality from Pandas and read generic .csv and .json files, respectively. They are highly customizable with optional parameters provided by Pandas read_csv and read_json functions. This means that anything you can load with these two functions, you can load into B-Store.

1.4.2 Code Examples for Custom Parsers/Readers

The following Jupyter notebooks demonstrate how to write custom Parsers and Readers.

1. https://github.com/kmdouglass/bstore/blob/master/examples/Tutorial%203%20-%20Writing%20custom%20parsers.ipynb
2. https://github.com/kmdouglass/bstore/blob/master/examples/Tutorial%204%20-%20Writing%20custom%20readers.ipynb

1.5 Using B-Store with Other Software

Author Kyle M. Douglass

Contact kyle.m.douglass@gmail.com

organization École Polytechnique Fédérale de Lausanne (EPFL)

revision $Revision: 0 $

date 2016-08-11

abstract B-Store uses the HDF file format, which means it may be used with other software programs for bio-image analysis.

Table of Contents

B-Store databases use the HDF file format for data storage. This means that any software package that can read from HDF files can also read B-Store databases.

1.5.1 ImageJ and Fiji

Widefield images found inside a B-Store database may be opened in ImageJ and Fiji using the HDF5 Plugin for ImageJ and Fiji. When using the GUI loader, use the option **individual hyperstacks (custom layout)** with `yz` as the **data set layout** argument.

This functionality requires that the image data in the HDF file possess an attribute called `element_size_um` that contains three floating point numbers corresponding to the size of a pixel in z, y, and x-directions. There are three ways that this attribute may be created when the database is built:

1. By specifying the **'HDFDatastore'_** `widefieldPixelSize` property, which is a two-element tuple of the x- and y- pixel sizes.
2. If `widefieldPixelSize` is None, the pixel size is extracted from the Micro-Manager metadata in the field specified by `__MM_PixelSize__` in B-Store's config.py.
3. If Micro-Manager metadata is not present, look for the pixel size in OME-XML metadata.
4. Failing this, the attribute `element_size_um` is not set.

If the images were not generated by Micro-Manager or a program that writes OME-XML metadata, simply specify the pixel size in the `widefieldPixelSize` attribute described above.

1.6 Frequently Asked Questions

Author Kyle M. Douglass

Contact kyle.m.douglass@gmail.com

organization École Polytechnique Fédérale de Lausanne (EPFL)

revision $Revision: 5 $

date 2017-01-24

abstract This document answers frequently asked questions regarding B-Store, a lightweight data management system for single molecule localization microscopy (SMLM).

Table of Contents

1.6.1 What is B-Store?

B-Store is a lightweight data management and analysis library for single molecule localization microscopy (SMLM). It serves two primary roles:

1. To structure SMLM data inside a single, high performance filetype for fast and easy information retrieval and storage.
2. To facilitate the analysis of high-throughput SMLM datasets.

What problem does B-Store solve?

High-throughput SMLM experiments can produce hundreds or even thousands of files containing multiple types of data (images, raw localizations, acquisition information, etc.). B-Store automatically sorts and stores this information in a datastore for rapid retrieval and analysis, removing any need to manually maintain the data yourself.

What are the design criteria for B-Store?

To realize these roles, B-Store is designed to meet these important criteria:

- Experimental datasets must be combined into a database-like structure that is easily readable by both humans and computers.

- Access and processing of data must be fast, regardless of the size of the dataset.
- Data provenance must be preserved throughout the organization and analysis pipeline.
- B-Store should not enforce standards that force scientists to adopt file formats, naming conventions, or software packages that differ from the ones they already use, except when it is absolutely necessary to achieve its roles.
- B-Store should be extensible to adapt to the changing needs of scientists using SMLM.
- Above all else, B-Store should make it easy to organize and document data and analysis pipelines to improve the reproducibility of SMLM experiments.

Of course, the changing needs of scientists means that B-Store will always be evolving to meet these criteria.

What doesn't B-Store do?

B-Store is efficient and fast because its scope is limited to SMLM data organization and analysis. In particular, B-Store does not:

- Calculate localizations from raw images.
- Control microscopy hardware.
- Provide database-like storage for core facilities.
- Generate any data or results for you. (Sorry.)

Why don't you use OME tools?

The Open Microscopy Environment (OME) is a wonderful set of software tools for working with bio-image data. In fact, the OME inspired this project in that B-Store emulates the OME model for archiving data, metadata, and analyses together in one abstract unit to improve reproducibility and communication of scientific results in SMLM.

In spite of this, we chose to develop tools independent of the OME for a few reasons. The OME was primarily designed for working with image data. SMLM data on the other hand is more heterogeneous than image data (localizations, drift correction, widefield images, etc.). Reworking parts of the OME to accomodate SMLM would therefore have been a significant undertaking on our part.

In addition, the OME database tool, OMERO, requires time for set up and maintenance. Many small labs doing SMLM may not be willing to invest the resources required for this. In contrast, B-Store is intended to be lightweight and require as little time for setup and maintenance as possible.

Some researchers in the SMLM community have expressed interest in extending the OME to SMLM, and we gladly welcome this effort. In the meanwhile, B-Store intends to satisfy the need for structured SMLM data.

1.6.2 How do I use B-Store?

B-Store is currently comprised of a set of functions, classes, and interfaces that are written in Python. You therefore can make B-Store datastores in any environment that runs Python code, including:

- The B-Store GUI
- Jupyter Notebooks
- IPython
- .py scripts

Once inside the datastore, the data may be accessed by any software that can read the HDF file format, including

1. B-Store

2. Python
3. MATLAB
4. ImageJ/Fiji
5. R
6. C/C++
7. Java

and more.

Is there a GUI interface?

There is currently a lightweight GUI interface for building HDF datastores.

Can I still use B-Store if I don't know Python?

If you don't know Python, you can still use B-Store in a number of ways.

The easiest way is to use the GUI. After that, try exploring the Jupyter notebooks in the examples folder. Find an example that does what you want, then modify the relevant parts, such as file names. Then, simply run the notebook.

You may also wish to use B-Store's datastore system, but not its analysis tools. In this case, you can use the notebooks to build your database, but access and analyze the data from the programming language of your choice, such as MATLAB. B-Store currently provides functionality for a datastore stored in an HDF file.

A third option is to call the Python code from within another language. Information for doing this in MATLAB may be found at the following link, though we have not yet tested this ourselves: http://www.mathworks.com/help/matlab/call-python-libraries.html

Of course, these approaches will only take you so far. Many parts of B-Store are meant to be customized to suit each scientist's needs, and these customizations are most easily implemented in Python. Regardless, the largest amount of customization you will want to do will likely be to write a Parser. A Parser converts raw acquisition and localization data into a format that can pass through the datastore interface. If your programming language can call Python and the HDFDatastore object, then you can write the parser in the language of your choice and then pass the parsed data through these interfaces to build your database.

1.6.3 How do I contribute to or extend B-Store?

B-Store was designed to be extensible. If you have an idea, code, or even a comment about how to improve it, we would love to hear about it!

A great place to start contributing is by posting questions or comments to the B-Store mailing list.

Common extensions you would want to do are to write plugins that extend the Parser, Processor, or Reader classes, or write your own DatasetTypes. If you add your custom Python files to the *~/.bstore/bsplugins* directory (*%USERPROFILE%\.bstore\bsplugins* on Windows), B-Store will know to search this directory for imports.

A custom Parser that we use in our own lab may be found here: https://github.com/kmdouglass/bsplugins-leb

How do I add my custom code to the B-Store project?

If you want to modify the B-Store code, you can start by forking the repository on GitHub. According to GitHub's documentation,

> A fork is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

After forking the repository, go ahead and make your changes, write some tests to be sure that your changes work like you expect them to, and then issue a pull request. The B-Store developers will review your suggested changes and, if they like them, will incorporate them into the B-Store project. With your permission your name will be added to the authors list.

For testing, B-Store uses the *nose* package. Type *nosetests* in the B-Store project root to run them. Test files are in the test files repository already mentioned. To run these successfully, set the *__Path_To_Test_Data__* variable in bstore/config.py.

What language is B-Store written in?

B-Store is written in the Python programming language (version 3) and relies heavily on a datatype known as a DataFrame. DataFrames and their functionality are provided by the Pandas library and in many ways work like Excel spreadsheets but are much, much faster. Pandas is highly optimized and used extensively for both normal and big data analytics at companies and research institutions across the globe.

In addition to Pandas, B-Store implements features provided by numerous scientific, open source Python libraries like numpy and matplotlib. If you can't do something in B-Store, you can likely still use these libraries to achieve what you want.

What is the logic of the B-Store datastore?

B-Store is designed to search specified directories on your computer for files associated with an SMLM experiment, such as those containing raw localizations and widefield images. These files are passed through a Parser, which converts them into a format suitable for insertion into a database. It does this by ensuring that the files satisfy the requirements of an interface known as a DatasetID. Data that implements this interface may pass into and out of the database; data that does not implement the interface cannot. You can think of the interface like a guard post at a government research facility. Only people with an ID badge for that facility (the interface) may enter. In principle, B-Store does not care about the data itself or the details of the database (HDF, SQL, etc.). At the moment, however, B-Store only supports databases contained in HDF files.

At the time this README file was written, the DatasetID of HDFDatastore consisted of the following properties:

- **acquisition ID** - integer identifying a specific acquisition
- **prefix** - a descriptive name for the acquisition, such as the cell type or condition
- **datasetType** - The type of data contained in the atom
- **attribute of** - For types that describe others, like localization metadata
- channel ID - the wavelength being imaged
- date ID - the date on which an acquisition was taken
- position ID - A single integer or integer pair identifying the position on the sample
- slice ID - An integer identifying the axial slice acquired
- replicate ID - An integer identifying the biological replicate that corresponds to this dataset.

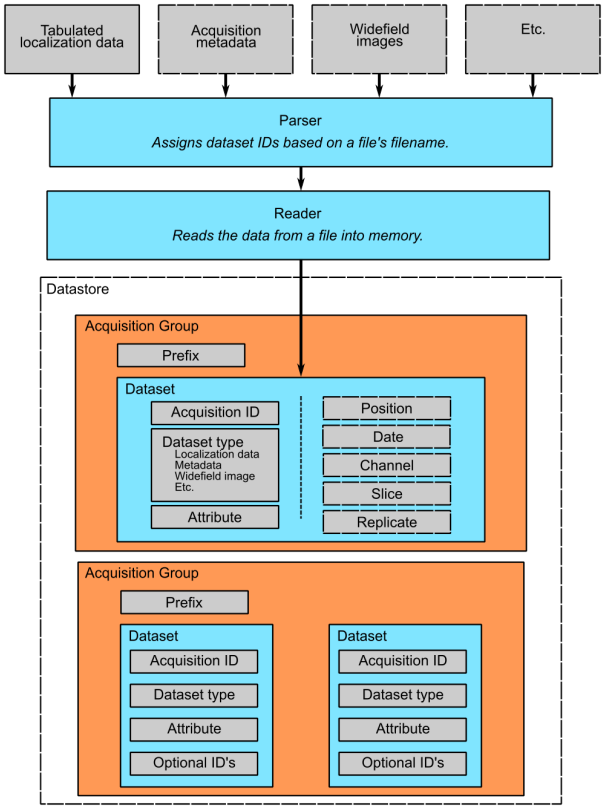
The first four properties in bold are required; the last properties are optional.

There are three important advantages to enforcing an interface such as this.

1. The computer will always know what kind of data it is working with and how to organize it.

2. The format of the data that you generate in your experiments can be made independent of the datastore, so you can do whatever you want to it. The Parser ensures that it is in the right format only at the point of datastore insertion.
3. The nature of the datastore and the types of data it can handle can grow and change in the future with minimal difficulty.

The logic of this interface is described graphically below. The raw data on top pass through the Parser and Readers objects and then into the datastore, where they are organized into acquisition groups. Each group is identified by a name called a prefix. Within the group, a dataset possesses an acquisition ID and a dataset type. An acqusition group is a set of datasets that were acquired during an experiment with the same prefix. A single dataset may optionally contain multiple fields of view (positions), wavelengths (channels), or axial slices. The database is therefore a collection of hierarchically arranged datasets, each belonging to a different acquisition group, and each uniquely identified by the conditions of the acquisition.



What is the logic behind the B-Store code?

The B-Store code base is divided into sixe separate modules:

1. parsers
2. database
3. readers
4. batch
5. processors
6. multiprocessors

In addition, functionality for each dataset type is specified in its own file in */bstore/datasetTypes/*.

The first three modules, parsers, database, and readers, contain all the code for organizing SMLM datasets into a datastore. The last three modules, batch, processors, and multiprocessors, are primarily used for extracting data from B-Store databases and performing (semi-)automated analyses.

Parsers

A parser reads files from a SMLM acquisition and produces a Dataset–an object that can be inserted into a B-Store datastore. This object will have mandatory and possibly optional fields for uniquely identifying the data within the datastore.

Database

The database module contains code for building datastores from raw data. It relies on a parser for translating files into a format that it knows how to work with.

Readers

Readers understand how to read data from files generated by different sources, such as ThunderSTORM or RapidSTORM, and convert them into a common and internal Python data type. This internal representation is temporary and is used to next write this data to HDF.

Readers were introduced in version 1.1.0 and lay the groundwork for a more customizable interface in later versions.

Batch

The batch module contains routines for performing automated analyses with B-Store databases. It allows you to build simple analysis pipelines for extracting just the data you need from the datastore.

Processors

Processors are objects that take just a few parameters. When called, they accept a single argument (usually a Pandas DataFrame) as an input and produce an object of the same datatype as an output with its data having been modified.

Examples of processors include common SMLM analysis steps such as Filter, Merge, and Cluster.

Multiprocessors

Multiprocessors are similar to processors. They differ in that they take multiple inputs to produce an output. One multiprocessor is called OverlayClusters, which overlays clusters of localizations onto a widefield image for visual inspection and anotation of cluster analyses.

What testing framework is used by the B-Store developers?

Unit tests for B-Store are written as functions with utilities provided by Python's nose package. Each module in B-Store has its own .py file containing these tests. They are stored in the bstore/tests and bstore/datasetTypes/tests folders in the B-Store root directory.

If you contribute to B-Store, we ask that you write unit tests for your code so that the developers can better understand what it's supposed to do before merging it into the main project.

1.6.4 Gotcha's

Spaces in column names

The library that B-Store uses to write to HDF files (PyTables) often has problems with spaces inside the names of DataFrame columns. We therefore recommend not using spaces. A workaround to this is to use the ConvertHeader processor to change column names during insertion into and retrieval from the database.

Widefield images

Grayscale

Widefield images are assumed to be grayscale. Unexpected behavior may result when attempting to place a color image into the database.

OME-XML

When reading metadata to determine the `element_size_um` attribute of the HDF `image_data`, the OME-XML metadata tags `PhysicalSizeX` and `PhysicalSizeY` will only be used if the corresponding units are in microns. This means the `PhysicalSizeXUnit` and `PhysicalSizeYUnit` must match the byte string `\xc2\xb5m`, which is UTF-8 for the Greek letter "mu", followed by the roman letter "m".

If Micro-Manager (MM) metadata with pixel size information is present, then the OME-XML data will be ignored in favor of the MM metadata.

See the page on using B-Store in other software packages for more information.

1.6.5 What is single molecule localization microscopy (SMLM)?

SMLM is a suite of super-resolution fluorescence microscopy techniques for imaging microscopic structures (like cells and organelles) with resolutions below the diffraction limit of light. A number of SMLM techniques exist, such as fPALM, PALM, STORM, and PAINT. In these microscopies, fluorescent molecules are made to "blink" on and off. A final image or dataset is computed by recording the positions of every blink for a period of time and adding together all the positions in the end.

SMLM is a powerful tool for helping scientists understand biology and chemistry at nanometer length scales. It is particularly well-suited for structural biology and for tracking single fluorescent molecules in time.

A fantastic movie explaining how this works using the blinking lights of the Eiffel tower was created by Ricardo Henriques. You can watch it here: https://www.youtube.com/watch?v=RE70GuMCzww

1.6.6 What does the "B" stand for?

"Blink"

1.7 Acknowledgments

Author Kyle M. Douglass

Contact kyle.m.douglass@gmail.com

organization École Polytechnique Fédérale de Lausanne (EPFL)

revision $Revision: 2 $

date 2016-11-06

abstract This is a list of people and organizations that made B-Store possible. Many, many thanks to everyone, even those who may not appear on this list, for their contributions great and small.

Table of Contents

1.7.1 Authors

- Kyle M. Douglass
- Niklas Berliner
- Marcel Stefko

1.7.2 People

- Suliana Manley
- Christian Sieben
- Aleksandra Vancevska

1.7.3 Organizations

- SystemsX.ch

1.7.4 Software

- Python
- Anaconda
- Pandas
- Jupyter
- trackpy
- NumPy

- SciPy
- matplotlib
- h5py
- scikit-learn
- tifffile
- filelock

CHAPTER 2

Programming

- modindex

CHAPTER 3

Misc.

- genindex
- search