Hangar

Release 0.5.2

Richard Izzo

Aug 03, 2020

CONTENTS

docs	
tests	
package	

Hangar is version control for tensor data. Commit, branch, merge, revert, and collaborate in the data-defined software era.

- Free software: Apache 2.0 license

CHAPTER

ONE

WHAT IS HANGAR?

Hangar is based off the belief that too much time is spent collecting, managing, and creating home-brewed version control systems for data. At it's core Hangar is designed to solve many of the same problems faced by traditional code version control system (ie. `Git`), just adapted for numerical data:

- Time travel through the historical evolution of a dataset.
- Zero-cost Branching to enable exploratory analysis and collaboration
- Cheap Merging to build datasets over time (with multiple collaborators)
- Completely abstracted organization and management of data files on disk
- Ability to only retrieve a small portion of the data (as needed) while still maintaining complete historical record
- Ability to push and pull changes directly to collaborators or a central server (ie a truly distributed version control system)

The ability of version control systems to perform these tasks for codebases is largely taken for granted by almost every developer today; However, we are in-fact standing on the shoulders of giants, with decades of engineering which has resulted in these phenomenally useful tools. Now that a new era of "Data-Defined software" is taking hold, we find there is a strong need for analogous version control systems which are designed to handle numerical data at large scale... Welcome to Hangar!

The Hangar Workflow:

```
   Checkout Branch
          |

 Create/Access Data
          |

Add/Remove/Update Samples
          |

       Commit
```

Log Style Output:

```
*   5254ec (master) : merge commit combining training updates and new validation␣
↪samples
|\
| * 650361 (add-validation-data) : Add validation labels and image data in isolated␣
↪branch
* | 5f15b4 : Add some metadata for later reference and add new training samples␣
↪received after initial import
|/
*   baddba : Initial commit adding training images and labels
```

Learn more about what Hangar is all about at https://hangar-py.readthedocs.io/

CHAPTER TWO

INSTALLATION

Hangar is in early alpha development release!

```
pip install hangar
```

CHAPTER

THREE

DOCUMENTATION

https://hangar-py.readthedocs.io/

CHAPTER

FOUR

DEVELOPMENT

To run the all tests run:

```
tox
```

Note, to combine the coverage data from all the tox environments run:

Windows	`set PYTEST_ADDOPTS=--cov-append`
	`tox`

Other	`PYTEST_ADDOPTS=--cov-append tox`

4.1 Overview

docs	
tests	
package	

Hangar is version control for tensor data. Commit, branch, merge, revert, and collaborate in the data-defined software era.

- Free software: Apache 2.0 license

4.1.1 What is Hangar?

Hangar is based off the belief that too much time is spent collecting, managing, and creating home-brewed version control systems for data. At it's core Hangar is designed to solve many of the same problems faced by traditional code version control system (ie. `Git`), just adapted for numerical data:

- Time travel through the historical evolution of a dataset.
- Zero-cost Branching to enable exploratory analysis and collaboration
- Cheap Merging to build datasets over time (with multiple collaborators)
- Completely abstracted organization and management of data files on disk
- Ability to only retrieve a small portion of the data (as needed) while still maintaining complete historical record
- Ability to push and pull changes directly to collaborators or a central server (ie a truly distributed version control system)

The ability of version control systems to perform these tasks for codebases is largely taken for granted by almost every developer today; However, we are in-fact standing on the shoulders of giants, with decades of engineering which has resulted in these phenomenally useful tools. Now that a new era of "Data-Defined software" is taking hold, we find there is a strong need for analogous version control systems which are designed to handle numerical data at large scale… Welcome to Hangar!

The Hangar Workflow:

```
   Checkout Branch
          |

 Create/Access Data
          |

Add/Remove/Update Samples
          |

       Commit
```

Log Style Output:

```
*   5254ec (master) : merge commit combining training updates and new validation␣
↪samples
|\
| * 650361 (add-validation-data) : Add validation labels and image data in isolated␣
↪branch
* | 5f15b4 : Add some metadata for later reference and add new training samples␣
↪received after initial import
|/
*   baddba : Initial commit adding training images and labels
```

Learn more about what Hangar is all about at https://hangar-py.readthedocs.io/

4.1.2 Installation

Hangar is in early alpha development release!

```
pip install hangar
```

4.1.3 Documentation

https://hangar-py.readthedocs.io/

4.1.4 Development

To run the all tests run:

```
tox
```

Note, to combine the coverage data from all the tox environments run:

Windows	`set PYTEST_ADDOPTS=--cov-append`
	`tox`

Other	`PYTEST_ADDOPTS=--cov-append tox`

4.2 Usage

To use Hangar in a project:

```
from hangar import Repository
```

Please refer to the *Hangar Tutorial* for examples, or *Hangar Core Concepts* to review the core concepts of the Hangar system.

4.3 Installation

For general usage it is recommended that you use a pre-built version of Hangar, either from a Python Distribution, or a pre-built wheel from PyPi.

4.3.1 Pre-Built Installation

Python Distributions

If you do not already use a Python Distribution, we recommend the Anaconda (or Miniconda) distribution, which supports all major operating systems (Windows, MacOSX, & the typical Linux variations). Detailed usage instructions are available on the anaconda website.

To install Hangar via the Anaconda Distribution (from the conda-forge conda channel):

```
conda install -c conda-forge hangar
```

Wheels (PyPi)

If you have an existing python installation on your computer, pre-built Hangar Wheels can be installed via pip from the Python Package Index (PyPi):

```
pip install hangar
```

4.3.2 Source Installation

To install Hangar from source, clone the repository from Github:

```
git clone https://github.com/tensorwerk/hangar-py.git
cd hangar-py
python setup.py install
```

Or use pip on the local package if you want to install all dependencies automatically in a development environment:

```
pip install -e .
```

Source installation in Google colab

Google colab comes with an older version of `h5py` pre-installed which is not compatible with hangar. If you need to install hangar from the source in google colab, make sure to uninstall the existing `h5py`

```
!pip uninstall h5py
```

Then follow the Source Installation steps given above.

4.4 Hangar Core Concepts

Warning: The usage info displayed in the `latest` build of the project documentation do not reflect recent changes to the API and internal structure of the project. They should not be relied on at the current moment; they will be updated over the next weeks, and will be in line before the next release.

This document provides a high level overview of the problems Hangar is designed to solve and introduces the core concepts for beginning to use Hangar.

4.4.1 What Is Hangar?

At its core Hangar is designed to solve many of the same problems faced by traditional code version control system (ie. `Git`), just adapted for numerical data:

- Time travel through the historical evolution of a dataset
- Zero-cost Branching to enable exploratory analysis and collaboration
- Cheap Merging to build datasets over time (with multiple collaborators)
- Completely abstracted organization and management of data files on disk
- Ability to only retrieve a small portion of the data (as needed) while still maintaining complete historical record
- Ability to push and pull changes directly to collaborators or a central server (ie. a truly distributed version control system)

The ability of version control systems to perform these tasks for codebases is largely taken for granted by almost every developer today; however, we are in-fact standing on the shoulders of giants, with decades of engineering which has resulted in these phenomenally useful tools. Now that a new era of "Data-Defined software" is taking hold, we find there is a strong need for analogous version control systems which are designed to handle numerical data at large scale… Welcome to Hangar!

4.4.2 Inspiration

The design of Hangar was heavily influenced by the Git source-code version control system. As a Hangar user, many of the fundamental building blocks and commands can be thought of as interchangeable:

- checkout
- commit
- branch
- merge
- diff
- push
- pull/fetch
- log

Emulating the high level the git syntax has allowed us to create a user experience which should be familiar in many ways to Hangar users; a goal of the project is to enable many of the same VCS workflows developers use for code while working with their data!

There are, however, many fundamental differences in how humans/programs interpret and use text in source files vs. numerical data which raise many questions Hangar needs to uniquely solve:

- How do we connect some piece of "Data" with a meaning in the real world?
- How do we diff and merge large collections of data samples?
- How can we resolve conflicts?
- How do we make data access (reading and writing) convenient for both user-driven exploratory analyses and high performance production systems operating without supervision?
- How can we enable people to work on huge datasets in a local (laptop grade) development environment?

We will show how Hangar solves these questions in a high-level guide below. For a deep dive into the Hangar internals, we invite you to check out the *Hangar Under The Hood* page.

4.4.3 How Hangar Thinks About Data

Abstraction 0: What is a Repository?

A “Repository” consists of an historically ordered mapping of “Commits” over time by various “Committers” across any number of “Branches”. Though there are many conceptual similarities in what a Git repo and a Hangar Repository achieve, Hangar is designed with the express purpose of dealing with numeric data. As such, when you read/write to/from a Repository, the main way of interaction with information will be through (an arbitrary number of) Columns in each Commit. A simple key/value store is also included to store metadata, but as it is a minor point is will largely be ignored for the rest of this post.

History exists at the Repository level, Information exists at the Commit level.

Abstraction 1: What is a Dataset?

Let’s get philosophical and talk about what a “Dataset” is. The word “Dataset” invokes some meaning to humans; a dataset may have a canonical name (like “MNIST” or “CoCo”), it will have a source where it comes from, (ideally) it has a purpose for some real-world task, it will have people who build, aggregate, and nurture it, and most importantly a Dataset always contains pieces of some type of information type which describes “something”.

It’s an abstract definition, but it is only us, the humans behind the machine, which associate “Data” with some meaning in the real world; it is in the same vein which we associate a group of Data in a “Dataset” with some real world meaning.

Our first abstraction is therefore the “Dataset”: a collection of (potentially groups of) data pieces observing a common form among instances which act to describe something meaningful. *To describe some phenomenon, a dataset may require multiple pieces of information, each of a particular format, for each instance/sample recorded in the dataset.*

> **For Example**
>
> a Hospital will typically have a *Dataset* containing all of the CT scans performed over some period of time. A single CT scan is an instance, a single sample; however, once many are grouped together they form a *Dataset*. To expand on this simple view we realize that each CT scan consists of hundreds of pieces of information:
>
> - Some large `numeric array` (the image data).
> - Some smaller `numeric tuples` (describing image spacing, dimension scale, capture time, machine parameters, etc).
> - Many pieces of `string` data (the patient name, doctor name, scan type, results found, etc).

When thinking about the group of CT scans in aggregate, we realize that though a single scan contains many disparate pieces of information stuck together, when thinking about the aggregation of every scan in the group, most of (if not all) of the same information fields are duplicated within each samples.

A single scan is a bunch of disparate information stuck together, many of those put together makes a Dataset, but looking down from the top, we identify pattern of common fields across all items. We call these groupings of similar typed information: **Columns**.

Abstraction 2: What Makes up a Column?

A Dataset is made of one or more Columns (and optionally some Metadata), with each item placed in some Column belonging to and making up an individual Sample. It is important to remember that all data needed to fully describe a single sample in a Dataset may consist of information spread across any number of Columns. To define a Column in Hangar, we only need to provide:

- a name
- a type
- a shape

The individual pieces of information (Data) which fully describe some phenomenon via an aggregate mapping access across any number of "Columns" are both individually and collectively referred to as Samples in the Hangar vernacular. According to the specification above, all samples contained in a Column must be numeric arrays with each having:

1) Same data type (standard numpy data types are supported).
2) A shape with each dimension size <= the shape (max shape) set in the column specification (more on this later).

Additionally, samples in a column can either be named, or unnamed (depending on how you interpret what the information contained in the column actually represents).

Effective use of Hangar relies on having an understanding of what exactly a "Sample" is in a particular Column. The most effective way to find out is to ask: "What is the smallest piece of data which has a useful meaning to 'me' (or 'my' downstream processes"). In the MNIST column, this would be a single digit image (a 28x28 array); for a medical column it might be an entire (512x320x320) MRI volume scan for a particular patient; while for the NASDAQ Stock Ticker it might be an hours worth of price data points (or less, or more!) The point is that **when you think about what a ``sample`` is, it should typically be the smallest atomic unit of useful information.**

Abstraction 3: What is Data?

From this point forward, **when we talk about "Data" we are actually talking about n-dimensional arrays of numeric information. To Hangar, "Data" is just a collection of numbers being passed into and out of it.** Data does not have a file type, it does not have a file-extension, it does not mean anything to Hangar itself - it is just numbers. This theory of "Data" is nearly as simple as it gets, and this simplicity is what enables us to be unconstrained as we build abstractions and utilities to operate on it.

Summary

```
A Dataset is thought of as containing Samples, but is actually defined by
Columns, which store parts of fully defined Samples in structures common
across the full aggregation of Dataset Samples.

This can essentially be represented as a key -> tensor mapping, which can
(optionally) be Sparse depending on usage patterns

                      Dataset
                         |
      -----------------------------------------
      |            |            |             |
   Column 1     Column 2     Column 3      Column 4
      |            |            |             |
------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
    image    |  filename  |   label    |  annotation |
------------------------------------------------------
      S1     |     S1     |            |      S1     |
      S2     |     S2     |     S2     |      S2     |
      S3     |     S3     |     S3     |             |
      S4     |     S4     |            |             |

More techincally, a Dataset is just a view over the columns that gives you
sample tuples based on the cross product of keys and columns. Hangar doesn't
store or track the data set, just the underlying columns.

 S1 = (image[S1], filename[S1], annotation[S1])
 S2 = (image[S2], filename[S2], label[S2], annotation[S2])
 S3 = (image[S3], filename[S3], label[S3])
 S4 = (image[S4], filename[S4])
```

Note: The technical crowd among the readers should note:

- Hangar preserves all sample data bit-exactly.
- Dense arrays are fully supported, Sparse array support is currently under development and will be released soon.
- Integrity checks are built in by default (explained in more detail in *Hangar Under The Hood*.) using cryptographically secure algorithms.
- Hangar is very much a young project, until penetration tests and security reviews are performed, we will refrain from stating that Hangar is fully "cryptographically secure". Security experts are welcome to contact us privately at hangar.info@tensorwerk.com to disclose any security issues.

4.4.4 Implications of the Hangar Data Philosophy

The Domain-Specific File Format Problem

Though it may seem counterintuitive at first, there is an incredible amount of freedom (and power) that is gained when "you" (the user) start to decouple some information container from the data which it actually holds. At the end of the day, the algorithms and systems you use to produce insight from data are just mathematical operations; math does not operate on a specific file type, math operates on numbers.

Human & Computational Cost

It seems strange that organizations & projects commonly rely on storing data on disk in some domain-specific - or custom built - binary format (ie. a `.jpg` image, `.nii` neuroimaging informatics study, `.cvs` tabular data, etc.), and just deal with the hassle of maintaining all the infrastructure around reading, writing, transforming, and preprocessing these files into useable numerical data every time they want to interact with their Columns. Even disregarding the computational cost/overhead of preprocessing & transforming the data on every read/write, these schemes require significant amounts of human capital (developer time) to be spent on building, testing, and upkeep/maintenance; all while adding significant complexity for users. Oh, and they also have a strangely high inclination to degenerate into horrible complexity which essentially becomes "magic" after the original creators move on.

The Hangar system is quite different in this regards. First, **we trust that you know what your data is and what it should be best represented as**. When writing to a Hangar repository, you process the data into n-dimensional arrays once. Then when you retrieve it you are provided with the same array, in the same shape and datatype (unless you ask for a particular subarray-slice), already initialized in memory and ready to compute on instantly.

High Performance From Simplicity

Because Hangar is designed to deal (almost exclusively) with numerical arrays, we are able to "stand on the shoulders of giants" once again by utilizing many of the well validated, highly optimized, and community validated numerical array data management utilities developed by the High Performance Computing community over the past few decades.

In a sense, the backend of Hangar serves two functions:

1) Bookkeeping: recording information about about columns, samples, commits, etc.
2) Data Storage: highly optimized interfaces which store and retrieve data from from disk through its backend utility.

The details are explained much more thoroughly in *Hangar Under The Hood*.

Because Hangar only considers data to be numbers, the choice of backend to store data is (in a sense) completely arbitrary so long as `Data In == Data Out`. **This fact has massive implications for the system**; instead of being tied to a single backend (each of which will have significant performance tradeoffs for arrays of particular datatypes, shapes, and access patterns), we simultaneously store different data pieces in the backend which is most suited to it. A great deal of care has been taken to optimize parameters in the backend interface which affects performance and compression of data samples.

The choice of backend to store a piece of data is selected automatically from heuristics based on the column specification, system details, and context of the storage service internal to Hangar. **As a user, this is completely transparent to you** in all steps of interacting with the repository. It does not require (or even accept) user specified configuration.

At the time of writing, Hangar has the following backends implemented (with plans to potentially support more as needs arise):

1) HDF5
2) Memmapped Arrays
3) TileDb (in development)

Open Source Software Style Collaboration in Dataset Curation

Specialized Domain Knowledge is A Scarce Resource

A common side effect of the *The Domain-Specific File Format Problem* is that anyone who wants to work with an organization's/project's data needs to not only have some domain expertise (so they can do useful things with the data), but they also need to have a non-trivial understanding of the projects dataset, file format, and access conventions / transformation pipelines. *In a world where highly specialized talent is already scarce, this phenomenon shrinks the pool of available collaborators dramatically.*

Given this situation, it's understandable why when most organizations spend massive amounts of money and time to build a team, collect & annotate data, and build an infrastructure around that information, they hold it for their private use with little regards for how the world could use it together. Businesses rely on proprietary information to stay ahead of their competitors, and because this information is so difficult (and expensive) to generate, it's completely reasonable that they should be the ones to benefit from all that work.

> **A Thought Experiment**
>
> Imagine that `Git` and `GitHub` didn't take over the world. Imagine that the `Diff` and `Patch` Unix tools never existed. Instead, imagine we were to live in a world where every software project had very different

> version control systems (largely homeade by non VCS experts, & not validated by a community over many years of use). Even worse, most of these tools don't allow users to easily branch, make changes, and automatically merge them back. It shouldn't be difficult to imagine how dramatically such a world would contrast to ours today. Open source software as we know it would hardly exist, and any efforts would probably be massively fragmented across the web (if there would even be a 'web' that we would recognize in this strange world).
>
> Without a way to collaborate in the open, open source software would largely not exist, and we would all be worse off for it.
>
> Doesn't this hypothetical sound quite a bit like the state of open source data collaboration in todays world?

The impetus for developing a tool like Hangar is the belief that if it is simple for anyone with domain knowledge to collaboratively curate columns containing information they care about, then they will.* Open source software development benefits everyone, we believe open source column curation can do the same.

How To Overcome The "Size" Problem

Even if the greatest tool imaginable existed to version, branch, and merge columns, it would face one massive problem which if it didn't solve would kill the project: *The size of data can very easily exceeds what can fit on (most) contributors laptops or personal workstations*. This section explains how Hangar can handle working with columns which are prohibitively large to download or store on a single machine.

As mentioned in *High Performance From Simplicity*, under the hood Hangar deals with "Data" and "Bookkeeping" completely separately. We've previously covered what exactly we mean by Data in *How Hangar Thinks About Data*, so we'll briefly cover the second major component of Hangar here. In short "Bookkeeping" describes everything about the repository. By everything, we do mean that the Bookkeeping records describe everything: all commits, parents, branches, columns, samples, data descriptors, schemas, commit message, etc. Though complete, these records are fairly small (tens of MB in size for decently sized repositories with decent history), and are highly compressed for fast transfer between a Hangar client/server.

> **A brief technical interlude**
>
> There is one very important (and rather complex) property which gives Hangar Bookeeping massive power: **Existence of some data piece is always known to Hangar and stored immutably once committed. However, the access pattern, backend, and locating information for this data piece may (and over time, will) be unique in every hangar repository instance**.
>
> Though the details of how this works is well beyond the scope of this document, the following example may provide some insight into the implications of this property:
>
> > If you `clone` some hangar repository, Bookeeping says that "some number of data pieces exist" and they should retrieved from the server. However, the bookeeping records transfered in a `fetch` / `push` / `clone` operation do not include information about where that piece of data existed on the client (or server) computer. Two synced repositories can use completely different backends to store the data, in completly different locations, and it does not matter - Hangar only guarantees that when collaborators ask for a data sample in some checkout, that they will be provided with identical arrays, not that they will come from the same place or be stored in the same way. Only when data is actually retrieved the "locating information" is set for that repository instance.

Because Hangar makes no assumptions about how/where it should retrieve some piece of data, or even an assumption that it exists on the local machine, and because records are small and completely describe history, once a machine has the Bookkeeping, it can decide what data it actually wants to materialize on it's local disk! These `partial fetch` / `partial clone` operations can materialize any desired data, whether it be for a few records at the head branch, for all data in a commit, or for the entire historical data. A future release will even include the ability to stream data directly to a Hangar checkout and materialize the data in memory without having to save it to disk at all!

More importantly: **Since Bookkeeping describes all history, merging can be performed between branches which may contain partial (or even no) actual data.** Aka **you don't need data on disk to merge changes into it.** It's an odd concept which will be explained more in depth in the future.

..note

```
To try this out for yourself, please refer to the the API Docs
(:ref:`ref-api`) on working with Remotes, especially the ``fetch()`` and
``fetch-data()`` methods. Otherwise look for through our tutorials &
examples for more practical info!
```

What Does it Mean to "Merge" Data?

We'll start this section, once again, with a comparison to source code version control systems. When dealing with source code text, merging is performed in order to take a set of changes made to a document, and logically insert the changes into some other version of the document. The goal is to generate a new version of the document with all changes made to it in a fashion which conforms to the "change author's" intentions. Simply put: the new version is valid and what is expected by the authors.

This concept of what it means to merge text does not generally map well to changes made in a column we'll explore why through this section, but look back to the philosophy of Data outlined in *How Hangar Thinks About Data* for inspiration as we begin. Remember, in the Hangar design a Sample is the smallest array which contains useful information. As any smaller selection of the sample array is meaningless, Hangar does not support subarray-slicing or per-index updates *when writing* data. (subarray-slice queries are permitted for read operations, though regular use is discouraged and may indicate that your samples are larger than they should be).

Diffing Hangar Checkouts

To understand merge logic, we first need to understand diffing, and the actors operations which can occur.

Addition An operation which creates a column, sample, or some metadata which did not previously exist in the relevant branch history.

Removal An operation which removes some column, a sample, or some metadata which existed in the parent of the commit under consideration. (Note: removing a column also removes all samples contained in it).

Mutation An operation which sets: data to a sample, the value of some metadata key, or a column schema, to a different value than what it had previously been created with (Note: a column schema mutation is observed when a column is removed, and a new column with the same name is created with a different dtype/shape, all in the same commit).

Merging Changes

Merging diffs solely consisting of additions and removals between branches is trivial, and performs exactly as one would expect from a text diff. Where things diverge from text is when we consider how we will merge diffs containing mutations.

Say we have some sample in commit A, a branch is created, the sample is updated, and commit C is created. At the same time, someone else checks out branch whose HEAD is at commit A, and commits a change to the sample as well. If these changes are identical, they are compatible, but what if they are not? In the following example, we diff and merge each element of the sample array like we would text:

```
                                               Merge ??
  commit A          commit B            Does combining mean anything?

[[0, 1, 2],        [[0, 1, 2],                [[1, 1, 1],
 [0, 1, 2], ----->  [2, 2, 2], ------------>  [2, 2, 2],
 [0, 1, 2]]         [3, 3, 3]]      /         [3, 3, 3]]
      \                            /
       \            commit C      /
        \                        /
         \          [[1, 1, 1], /
          ------->   [0, 1, 2],
                     [0, 1, 2]]
```

We see that a result can be generated, and can agree if this was a piece of text, the result would be correct. Don't be fooled, this is an abomination and utterly wrong/meaningless. Remember we said earlier `"the result of a merge should conform to the intentions of each author"`. This merge result conforms to neither author's intention. The value of an array element is not isolated, every value affects how the entire sample is understood. The values at Commit B or commit C may be fine on their own, but if two samples are mutated independently with non-identical updates, it is a conflict that needs to be handled by the authors.

This is the actual behavior of Hangar.

```
  commit A          commit B

[[0, 1, 2],        [[0, 1, 2],
 [0, 1, 2], ----->  [2, 2, 2], ----- MERGE CONFLICT
 [0, 1, 2]]         [3, 3, 3]]      /
      \                            /
       \            commit C      /
        \                        /
         \          [[1, 1, 1], /
          ------->   [0, 1, 2],
                     [0, 1, 2]]
```

When a conflict is detected, the merge author must either pick a sample from one of the commits or make changes in one of the branches such that the conflicting sample values are resolved.

How Are Conflicts Detected?

Any merge conflicts can be identified and addressed ahead of running a `merge` command by using the built in `diff` tools. When diffing commits, Hangar will provide a list of conflicts which it identifies. In general these fall into 4 categories:

1) **Additions** in both branches which created new keys (samples / columns / metadata) with non-compatible values. For samples & metadata, the hash of the data is compared, for columns, the schema specification is checked for compatibility in a method custom to the internal workings of Hangar.

2) **Removal** in `Master Commit / Branch` **& Mutation** in `Dev Commit / Branch`. Applies for samples, columns, and metadata identically.

3) **Mutation** in `Dev Commit / Branch` **& Removal** in `Master Commit / Branch`. Applies for samples, columns, and metadata identically.

4) **Mutations** on keys both branches to non-compatible values. For samples & metadata, the hash of the data is compared, for columns, the schema specification is checked for compatibility in a method custom to the internal workings of Hangar.

4.4.5 What's Next?

- Get started using Hangar today: *Installation*.
- Read the tutorials: *Hangar Tutorial*.
- Dive into the details: *Hangar Under The Hood*.

4.5 Python API

This is the python API for the Hangar project.

4.5.1 Repository

class Repository(*path: Union[str, pathlib.Path], exists: bool = True*)

Launching point for all user operations in a Hangar repository.

All interaction, including the ability to initialize a repo, checkout a commit (for either reading or writing), create a branch, merge branches, or generally view the contents or state of the local repository starts here. Just provide this class instance with a path to an existing Hangar repository, or to a directory one should be initialized, and all required data for starting your work on the repo will automatically be populated.

```
>>> from hangar import Repository
>>> repo = Repository('foo/path/to/dir')
```

Parameters

- **path** (*Union[str, os.PathLike]*) – local directory path where the Hangar repository exists (or initialized)
- **exists** (*bool, optional*) – True if a Hangar repository should exist at the given directory path. Should no Hangar repository exists at that location, a UserWarning will be raised indicating that the `init()` method needs to be called.

  False if the provided path does not need to (but optionally can) contain a Hangar repository. if a Hangar repository does not exist at that path, the usual UserWarning will be suppressed.

  In both cases, the path must exist and the user must have sufficient OS permissions to write to that location. Default = True

checkout(*write: bool = False, *, branch: str = '', commit: str = ''*) → Union[*hangar.checkout.ReaderCheckout*, *hangar.checkout.WriterCheckout*]

Checkout the repo at some point in time in either *read* or *write* mode.

Only one writer instance can exist at a time. Write enabled checkout must must create a staging area from the `HEAD` commit of a branch. On the contrary, any number of reader checkouts can exist at the same time and can specify either a branch name or a commit hash.

Parameters

- **write** (*bool, optional*) – Specify if the checkout is write capable, defaults to False
- **branch** (*str, optional*) – name of the branch to checkout. This utilizes the state of the repo as it existed at the branch `HEAD` commit when this checkout object was instantiated, defaults to ‘’

- **commit** (*str, optional*) – specific hash of a commit to use for the checkout (instead of a branch HEAD commit). This argument takes precedent over a branch name parameter if it is set. Note: this only will be used in non-writeable checkouts, defaults to ‘’

Raises

- **ValueError** – If the value of *write* argument is not boolean
- **ValueError** – If `commit` argument is set to any value when `write=True`. Only `branch` argument is allowed.

Returns Checkout object which can be used to interact with the repository data

Return type Union[*ReaderCheckout*, *WriterCheckout*]

clone (*user_name: str*, *user_email: str*, *remote_address: str*, *, *remove_old: bool = False*) → str

Download a remote repository to the local disk.

The clone method implemented here is very similar to a *git clone* operation. This method will pull all commit records, history, and data which are parents of the remote’s *master* branch head commit. If a `Repository` exists at the specified directory, the operation will fail.

Parameters

- **user_name** (*str*) – Name of the person who will make commits to the repository. This information is recorded permanently in the commit records.
- **user_email** (*str*) – Email address of the repository user. This information is recorded permanently in any commits created.
- **remote_address** (*str*) – location where the `hangar.remote.server.HangarServer` process is running and accessible by the clone user.
- **remove_old** (*bool, optional, kwarg only*) – DANGER! DEVELOPMENT USE ONLY! If enabled, a `hangar.repository.Repository` existing on disk at the same path as the requested clone location will be completely removed and replaced with the newly cloned repo. (the default is False, which will not modify any contents on disk and which will refuse to create a repository at a given location if one already exists there.)

Returns Name of the master branch for the newly cloned repository.

Return type str

create_branch (*name: str*, *base_commit: str = None*) → hangar.records.heads.BranchHead

create a branch with the provided name from a certain commit.

If no base commit hash is specified, the current writer branch `HEAD` commit is used as the `base_commit` hash for the branch. Note that creating a branch does not actually create a checkout object for interaction with the data. to interact you must use the repository checkout method to properly initialize a read (or write) enabled checkout object.

```
>>> from hangar import Repository
>>> repo = Repository('foo/path/to/dir')
```

```
>>> repo.create_branch('testbranch')
    BranchHead(name='testbranch', digest='b66b...a8cc')
>>> repo.list_branches()
    ['master', 'testbranch']
>>> co = repo.checkout(write=True, branch='testbranch')
>>> # add data ...
>>> newDigest = co.commit('added some stuff')
>>> repo.create_branch('new-changes', base_commit=newDigest)
    BranchHead(name='new-changes', digest='35kd...3254')
>>> repo.list_branches()
    ['master', 'new-changes', 'testbranch']
```

Parameters

- **name** (*str*) – name to assign to the new branch
- **base_commit** (*str, optional*) – commit hash to start the branch root at. if not specified, the writer branch `HEAD` commit at the time of execution will be used, defaults to None

Returns NamedTuple[str, str] with fields for `name` and `digest` of the branch created (if the operation was successful)

Return type `BranchHead`

Raises

- **ValueError** – If the branch name provided contains characters outside of alpha-numeric ascii characters and ".", "_", "-" (no whitespace), or is > 64 characters.
- **ValueError** – If the branch already exists.
- **RuntimeError** – If the repository does not have at-least one commit on the "default" (ie. `master`) branch.

diff(*master: str, dev: str*) → hangar.diff.DiffAndConflicts

Calculate diff between master and dev branch/commits.

Diff is calculated as if we are to merge "dev" into "master"

Parameters

- **master** (*str*) – branch name or commit hash digest to use as the "master" which changes made in "dev" are compared to.
- **dev** (*str*) – branch name or commit hash digest to use as the "dev" (ie. "feature") branch which changes have been made to which are to be compared to the contents of "master".

Returns Standard output diff structure.

Return type DiffAndConflicts

force_release_writer_lock() → bool

Force release the lock left behind by an unclosed writer-checkout

Warning: *NEVER USE THIS METHOD IF WRITER PROCESS IS CURRENTLY ACTIVE.* At the time of writing, the implications of improper/malicious use of this are not understood, and there is a a risk of of undefined behavior or (potentially) data corruption.

At the moment, the responsibility to close a write-enabled checkout is placed entirely on the user. If the *close()* method is not called before the program terminates, a new checkout with write=True will fail. The lock can only be released via a call to this method.

Note: This entire mechanism is subject to review/replacement in the future.

Returns if the operation was successful.

Return type bool

init(*user_name: str*, *user_email: str*, ***, *remove_old: bool = False*) → str

Initialize a Hangar repository at the specified directory path.

This function must be called before a checkout can be performed.

Parameters

- **user_name** (*str*) – Name of the repository user account.
- **user_email** (*str*) – Email address of the repository user account.
- **remove_old** (*bool, kwarg-only*) – DEVELOPER USE ONLY – remove and reinitialize a Hangar repository at the given path, Default = False

Returns the full directory path where the Hangar repository was initialized on disk.

Return type str

property initialized

Check if the repository has been initialized or not

Returns True if repository has been initialized.

Return type bool

list_branches() → List[str]

list all branch names created in the repository.

Returns the branch names recorded in the repository

Return type List[str]

log(*branch: str = None*, *commit: str = None*, ***, *return_contents: bool = False*, *show_time: bool = False*, *show_user: bool = False*) → Optional[dict]

Displays a pretty printed commit log graph to the terminal.

Note: For programatic access, the return_contents value can be set to true which will retrieve relevant commit specifications as dictionary elements.

Parameters

- **branch** (*str, optional*) – The name of the branch to start the log process from. (Default value = None)
- **commit** (*str, optional*) – The commit hash to start the log process from. (Default value = None)
- **return_contents** (*bool, optional, kwarg only*) – If true, return the commit graph specifications in a dictionary suitable for programatic access/evaluation.
- **show_time** (*bool, optional, kwarg only*) – If true and return_contents is False, show the time of each commit on the printed log graph
- **show_user** (*bool, optional, kwarg only*) – If true and return_contents is False, show the committer of each commit on the printed log graph

Returns Dict containing the commit ancestor graph, and all specifications.

Return type Optional[dict]

merge (*message: str*, *master_branch: str*, *dev_branch: str*) → str

Perform a merge of the changes made on two branches.

Parameters

- **message** (*str*) – Commit message to use for this merge.
- **master_branch** (*str*) – name of the master branch to merge into
- **dev_branch** (*str*) – name of the dev/feature branch to merge

Returns Hash of the commit which is written if possible.

Return type str

property path

Return the path to the repository on disk, read-only attribute

Returns path to the specified repository, not including *.hangar* directory

Return type str

property remote

Accessor to the methods controlling remote interactions.

See also:

Remotes for available methods of this property

Returns Accessor object methods for controlling remote interactions.

Return type *Remotes*

remove_branch (*name: str*, *, *force_delete: bool = False*) → hangar.records.heads.BranchHead

Permanently delete a branch pointer from the repository history.

Since a branch (by definition) is the name associated with the HEAD commit of a historical path, the default behavior of this method is to throw an exception (no-op) should the HEAD not be referenced as an ancestor (or at least as a twin) of a separate branch which is currently *ALIVE*. If referenced in another branch's history, we are assured that all changes have been merged and recorded, and that this pointer can be safely deleted without risk of damage to historical provenance or (eventual) loss to garbage collection.

```
>>> from hangar import Repository
>>> repo = Repository('foo/path/to/dir')
```

```
>>> repo.create_branch('first-testbranch')
BranchHead(name='first-testbranch', digest='9785...56da')
>>> repo.create_branch('second-testbranch')
BranchHead(name='second-testbranch', digest='9785...56da')
>>> repo.list_branches()
['master', 'first-testbranch', 'second-testbranch']
>>> # Make a commit to advance a branch
>>> co = repo.checkout(write=True, branch='first-testbranch')
>>> # add data ...
>>> co.commit('added some stuff')
'3l253la5hna3k3a553256nak35hq5q534kq35532'
>>> co.close()
```

```
>>> repo.remove_branch('second-testbranch')
BranchHead(name='second-testbranch', digest='9785...56da')
```

A user may manually specify to delete an un-merged branch, in which case the `force_delete` keyword-only argument should be set to `True`.

```
>>> # check out master and try to remove 'first-testbranch'
>>> co = repo.checkout(write=True, branch='master')
>>> co.close()
```

```
>>> repo.remove_branch('first-testbranch')
Traceback (most recent call last):
    ...
RuntimeError: ("The branch first-testbranch is not fully merged. "
"If you are sure you want to delete it, re-run with "
"force-remove parameter set.")
>>> # Now set the `force_delete` parameter
>>> repo.remove_branch('first-testbranch', force_delete=True)
BranchHead(name='first-testbranch', digest='9785...56da')
```

It is important to note that *while this method will handle all safety checks, argument validation, and performs the operation to permanently delete a branch name/digest pointer, **no commit refs along the history will be deleted from the Hangar database*.* Most of the history contains commit refs which must be safe in other branch histories, and recent commits may have been used as the base for some new history. As such, even if some of the latest commits leading up to a deleted branch `HEAD` are orphaned (unreachable), the records (and all data added in those commits) will remain on the disk.

In the future, we intend to implement a garbage collector which will remove orphan commits which have not been modified for some set amount of time (probably on the order of a few months), but this is not implemented at the moment.

Should an accidental forced branch deletion occur, *it is possible to recover* and create a new branch head pointing to the same commit. If the commit digest of the removed branch `HEAD` is known, its as simple as specifying a name and the `base_digest` in the normal *`create_branch()`* method. If the digest is unknown, it will be a bit more work, but some of the developer facing introspection tools / routines could be used to either manually or (with minimal effort) programmatically find the orphan commit candidates. If you find yourself having accidentally deleted a branch, and must get it back, please reach out on the Github Issues page. We'll gladly explain more in depth and walk you through the process in any way we can help!

Parameters

- **name** (*str*) – name of the branch which should be deleted. This branch must exist, and cannot refer to a remote tracked branch (ie. origin/devbranch), please see exception descriptions for other parameters determining validity of argument
- **force_delete** (*bool, optional*) – If True, remove the branch pointer even if the changes are un-merged in other branch histories. May result in orphaned commits which may be time-consuming to recover if needed, by default False

Returns NamedTuple[str, str] with fields for *name* and *digest* of the branch pointer deleted.

Return type `BranchHead`

Raises

- **ValueError** – If a branch with the provided name does not exist locally
- **PermissionError** – If removal of the branch would result in a repository with zero local branches.
- **PermissionError** – If a write enabled checkout is holding the writer-lock at time of this call.

- **PermissionError** – If the branch to be removed was the last used in a write-enabled checkout, and whose contents form the base of the staging area.
- **RuntimeError** – If the branch has not been fully merged into other branch histories, and `force_delete` option is not `True`.

property size_human

Disk space used by the repository returned in human readable string.

```
>>> repo.size_human
'1.23 GB'
>>> print(type(repo.size_human))
<class 'str'>
```

Returns disk space used by the repository formated in human readable text.

Return type str

property size_nbytes

Disk space used by the repository returned in number of bytes.

```
>>> repo.size_nbytes
1234567890
>>> print(type(repo.size_nbytes))
<class 'int'>
```

Returns number of bytes used by the repository on disk.

Return type int

summary(*, *branch: str = ''*, *commit: str = ''*) → None

Print a summary of the repository contents to the terminal

Parameters

- **branch** (*str, optional*) – A specific branch name whose head commit will be used as the summary point (Default value = ‘’)
- **commit** (*str, optional*) – A specific commit hash which should be used as the summary point. (Default value = ‘’)

verify_repo_integrity() → bool

Verify the integrity of the repository data on disk.

Runs a full cryptographic verification of repository contents in order to ensure the integrity of all data and history recorded on disk.

Note: This proof may take a significant amount of time to run for repositories which:

1. store significant quantities of data on disk.
2. have a very large number of commits in their history.

As a brief explanation for why these are the driving factors behind processing time:

1. Every single piece of data in the repositories history must be read from disk, cryptographically hashed, and compared to the expected value. There is no exception to this rule; regardless of when a piece of data was added / removed from an column, or for how many (or how few) commits some sample exists in. The integrity of the commit tree at any point after some piece of data is added to the repo can only be validated if it - and all earlier data pieces - are proven to be intact and unchanged.

Note: This does not mean that the verification is repeatedly performed for every commit some piece of data is stored in. Each data piece is read from disk and verified only once, regardless of how many commits some piece of data is referenced in.

2. Each commit reference (defining names / contents of a commit) must be decompressed and parsed into a usable data structure. We scan across all data digests referenced in the commit and ensure that the corresponding data piece is known to hangar (and validated as unchanged). The commit refs (along with the corresponding user records, message, and parent map), are then re-serialized and cryptographically hashed for comparison to the expected value. While this process is fairly efficient for a single commit, it must be repeated for each commit in the repository history, and may take a non-trivial amount of time for repositories with thousands of commits.

While the two points above are the most time consuming operations, there are many more checks which are performed alongside them as part of the full verification run.

Returns True if integrity verification is successful, otherwise False; in this case, a message describing the offending component will be printed to stdout.

Return type bool

property version

Find the version of Hangar software the repository is written with

Returns semantic version of major, minor, micro version of repo software version.

Return type str

property writer_lock_held

Check if the writer lock is currently marked as held. Read-only attribute.

Returns True is writer-lock is held, False if writer-lock is free.

Return type bool

4.5.2 Remotes

class Remotes

Class which governs access to remote interactor objects.

Note: The remote-server implementation is under heavy development, and is likely to undergo changes in the Future. While we intend to ensure compatability between software versions of Hangar repositories written to disk, the API is likely to change. Please follow our process at: https://www.github.com/tensorwerk/hangar-py

add(*name: str*, *address: str*) → hangar.remotes.RemoteInfo

Add a remote to the repository accessible by *name* at *address*.

Parameters

- **name** (`str`) – the name which should be used to refer to the remote server (ie: 'origin')
- **address** (`str`) – the IP:PORT where the hangar server is running

Returns Two-tuple containing (`name`, `address`) of the remote added to the client's server list.

Return type RemoteInfo

Raises

- **ValueError** – If provided name contains any non ascii letter characters characters, or if the string is longer than 64 characters long.

- **ValueError** – If a remote with the provided name is already listed on this client, No-Op. In order to update a remote server address, it must be removed and then re-added with the desired address.

fetch(*remote: str*, *branch: str*) → str

Retrieve new commits made on a remote repository branch.

This is semantically identical to a *git fetch* command. Any new commits along the branch will be retrieved, but placed on an isolated branch to the local copy (ie. `remote_name/branch_name`). In order to unify histories, simply merge the remote branch into the local branch.

Parameters

- **remote** (*str*) – name of the remote repository to fetch from (ie. `origin`)
- **branch** (*str*) – name of the branch to fetch the commit references for.

Returns Name of the branch which stores the retrieved commits.

Return type str

fetch_data(*remote: str*, *branch: str = None*, *commit: str = None*, ***, *column_names: Optional[Sequence[str]] = None*, *max_num_bytes: int = None*, *retrieve_all_history: bool = False*) → List[str]

Retrieve the data for some commit which exists in a *partial* state.

Parameters

- **remote** (*str*) – name of the remote to pull the data from
- **branch** (*str, optional*) – The name of a branch whose HEAD will be used as the data fetch point. If None, `commit` argument expected, by default None
- **commit** (*str, optional*) – Commit hash to retrieve data for, If None, `branch` argument expected, by default None
- **column_names** (*Optional[Sequence[str]]*) – Names of the columns which should be retrieved for the particular commits, any columns not named will not have their data fetched from the server. Default behavior is to retrieve all columns
- **max_num_bytes** (*Optional[int]*) – If you wish to limit the amount of data sent to the local machine, set a *max_num_bytes* parameter. This will retrieve only this amount of data from the server to be placed on the local disk. Default is to retrieve all data regardless of how large.
- **retrieve_all_history** (*Optional[bool]*) – if data should be retrieved for all history accessible by the parents of this commit HEAD. by default False

Returns commit hashes of the data which was returned.

Return type List[str]

Raises

- **ValueError** – if branch and commit args are set simultaneously.
- **ValueError** – if specified commit does not exist in the repository.
- **ValueError** – if branch name does not exist in the repository.

fetch_data_sample(*remote: str*, *column: str*, *samples: Union[str, int, Sequence[Union[str, int]], Sequence[Union[Tuple[Union[str, int], Union[str, int]], Tuple[Union[str, int]], str, int]]]*, *branch: Optional[str] = None*, *commit: Optional[str] = None*) → str

Granular fetch data operation allowing selection of individual samples.

> **Warning:** This is a specialized version of the `fetch_data()` method for use in specilized situations where some prior knowledge is known about the data. Most users should prefer `fetch_data()` over this version.

In some cases, it may be desireable to only perform a fetch data operation for some particular samples within a column (without needing to download any other data contained in the column). This method allows for the granular specification of keys to fetch in a certain column at the selected *branch* / *commit* time point.

Parameters

- **remote** (*str*) – name of the remote server to pull data from
- **column** (*str*) – name of the column which data is being fetched from.
- **samples** (*Union[KeyType, Sequence[KeyType],)* – Sequence[Union[Tuple[KeyType, KeyType], Tuple[KeyType], KeyType]]] Key, or sequence of sample keys to select.
  - Flat column layouts should provide just a single key, or flat sequence of keys which will be fetched from the server. ie. *sample1* OR [*sample1*, *sample2*, *sample3*, etc.]
  - Nested column layouts can provide tuples specifying *(sample, subsample)* records to retrieve, tuples with an *Ellipsis* character in the *subsample* index *(sample, ...)* (which will fetch all subsamples for the given sample), or can provide lone sample keys in the sequences *sample* (which will also fetch all subsamples listed under the sample) OR ANY COMBINATION of the above.
- **branch** (*Optional[str]*) – branch head to operate on, either `branch` or `commit` argument must be passed, but NOT both. Default is `None`
- **commit** (*Optional[str]*) – commit to operate on, either *branch* or *commit* argument must be passed, but NOT both.

Returns On success, the commit hash which data was fetched into.

Return type str

list_all() → List[hangar.remotes.RemoteInfo]

List all remote names and addresses recorded in the client's repository.

Returns list of namedtuple specifying (`name`, `address`) for each remote server recorded in the client repo.

Return type List[RemoteInfo]

ping(*name: str*) → float

Ping remote server and check the round trip time.

Parameters **name** (*str*) – name of the remote server to ping

Returns round trip time it took to ping the server after the connection was established and requested client configuration was retrieved

Return type float

Raises

- **KeyError** – If no remote with the provided name is recorded.
- **ConnectionError** – If the remote server could not be reached.

push(*remote: str*, *branch: str*, ***, *username: str = ''*, *password: str = ''*) → str

push changes made on a local repository to a remote repository.

This method is semantically identical to a `git push` operation. Any local updates will be sent to the remote repository.

Note: The current implementation is not capable of performing a `force push` operation. As such, remote branches with diverged histories to the local repo must be retrieved, locally merged, then re-pushed. This feature will be added in the near future.

Parameters

- **remote** (*str*) – name of the remote repository to make the push on.
- **branch** (*str*) – Name of the branch to push to the remote. If the branch name does not exist on the remote, the it will be created
- **username** (*str, optional, kwarg-only*) – credentials to use for authentication if repository push restrictions are enabled, by default ‘’.
- **password** (*str, optional, kwarg-only*) – credentials to use for authentication if repository push restrictions are enabled, by default ‘’.

Returns Name of the branch which was pushed

Return type str

remove(*name: str*) → hangar.remotes.RemoteInfo

Remove a remote repository from the branch records

Parameters **name** (*str*) – name of the remote to remove the reference to

Raises **KeyError** – If a remote with the provided name does not exist

Returns The channel address which was removed at the given remote name

Return type str

4.5.3 Write Enabled Checkout

Checkout

class WriterCheckout

Checkout the repository at the head of a given branch for writing.

This is the entry point for all writing operations to the repository, the writer class records all interactions in a special `"staging"` area, which is based off the state of the repository as it existed at the `HEAD` commit of a branch.

```
>>> co = repo.checkout(write=True)
>>> co.branch_name
'master'
>>> co.commit_hash
'masterheadcommithash'
>>> co.close()
```

At the moment, only one instance of this class can write data to the staging area at a time. After the desired operations have been completed, it is crucial to call `close()` to release the writer lock. In addition, after any changes have been made to the staging area, the branch `HEAD` cannot be changed. In order to checkout another branch `HEAD` for writing, you must either `commit()` the changes, or perform a hard-reset of the staging area to the last commit via `reset_staging_area()`.

In order to reduce the chance that the python interpreter is shut down without calling `close()`, which releases the writer lock - a common mistake during ipython / jupyter sessions - an atexit hook is registered to `close()`. If properly closed by the user, the hook is unregistered after completion with no ill effects. So long as a the process is NOT terminated via non-python SIGKILL, fatal internal python error, or or special os exit methods, cleanup will occur on interpreter shutdown and the writer lock will be released. If a non-handled termination method does occur, the `force_release_writer_lock()` method must be called manually when a new python process wishes to open the writer checkout.

`__contains__`(*key*)

Determine if some column name (key) exists in the checkout.

`__getitem__`(*index*)

Dictionary style access to columns and samples

Checkout object can be thought of as a "dataset" ("dset") mapping a view of samples across columns.

```
>>> dset = repo.checkout(branch='master')
>>>
# Get an column contained in the checkout.
>>> dset['foo']
ColumnDataReader
>>>
# Get a specific sample from ``'foo'`` (returns a single array)
>>> dset['foo', '1']
np.array([1])
>>>
# Get multiple samples from ``'foo'`` (returns a list of arrays, in order
# of input keys)
>>> dset[['foo', '1'], ['foo', '2'],  ['foo', '324']]
[np.array([1]), np.ndarray([2]), np.ndarray([324])]
>>>
# Get sample from multiple columns, column/data returned is ordered
# in same manner as input of func.
>>> dset[['foo', '1'], ['bar', '1'],  ['baz', '1']]
[np.array([1]), np.ndarray([1, 1]), np.ndarray([1, 1, 1])]
>>>
# Get multiple samples from multiple columns            >>> keys = [(col,␣
↪str(samp)) for samp in range(2) for col in ['foo', 'bar']]
>>> keys
[('foo', '0'), ('bar', '0'), ('foo', '1'), ('bar', '1')]
>>> dset[keys]
[np.array([1]), np.array([1, 1]), np.array([2]), np.array([2, 2])]
```

Arbitrary column layouts are supported by simply adding additional members to the keys for each piece of data. For example, getting data from a column with a nested layout:

```
>>> dset['nested_col', 'sample_1', 'subsample_0']
np.array([1, 0])
>>>
# a sample accessor object can be retrieved at will...
>>> dset['nested_col', 'sample_1']
<class 'FlatSubsampleReader'>(column_name='nested_col', sample_name='sample_1
↪')
```

(continues on next page)

(continued from previous page)

```
>>>
# to get all subsamples in a nested sample use the Ellipsis operator
>>> dset['nested_col', 'sample_1', ...]
{'subsample_0': np.array([1, 0]),
 'subsample_1': np.array([1, 1]),
 ...
 'subsample_n': np.array([1, 255])}
```

Retrieval of data from different column types can be mixed and combined as desired. For example, retrieving data from both flat and nested columns simultaneously:

```
>>> dset[('nested_col', 'sample_1', '0'), ('foo', '0')]
[np.array([1, 0]), np.array([0])]
>>> dset[('nested_col', 'sample_1', ...), ('foo', '0')]
[{'subsample_0': np.array([1, 0]), 'subsample_1': np.array([1, 1])},
 np.array([0])]
>>> dset[('foo', '0'), ('nested_col', 'sample_1')]
[np.array([0]),
 <class 'FlatSubsampleReader'>(column_name='nested_col', sample_name='sample_1
↪')]
```

If a column or data key does not exist, then this method will raise a KeyError. As an alternative, missing keys can be gracefully handeled by calling `get()` instead. This method does not (by default) raise an error if a key is missing. Instead, a (configurable) default value is simply inserted in it's place.

```
>>> dset['foo', 'DOES_NOT_EXIST']
-------------------------------------------------------------------
KeyError                           Traceback (most recent call last)
<ipython-input-40-731e6ea62fb8> in <module>
----> 1 res = co['foo', 'DOES_NOT_EXIST']
KeyError: 'DOES_NOT_EXIST'
```

Parameters **`index`** – column name, sample key(s) or sequence of list/tuple of column name, sample keys(s) which should be retrieved in the operation.

Please see detailed explanation above for full explanation of accepted argument format / result types.

Returns

- `Columns` – single column parameter, no samples specified
- *Any* – Single column specified, single sample key specified
- *List[Any]* – arbitrary columns, multiple samples array data for each sample is returned in same order sample keys are received.

`__iter__`()

Iterate over column keys

`__len__`()

Returns number of columns in the checkout.

`add_bytes_column`(*name: str, contains_subsamples: bool = False, *, backend: Optional[str] = None, backend_options: Optional[dict] = None*)

Initializes a `bytes` container column

Columns are created in order to store some arbitrary collection of data pieces. In this case, we store `bbytes` data. Items need not be related to each-other in any direct capacity; the only criteria hangar requires is that all pieces of data stored in the column have a compatible schema with each-other (more on this below). Each piece of data is indexed by some key (either user defined or automatically generated depending on the user's preferences). Both single level stores (sample keys mapping to data on disk) and nested stores (where some sample key maps to an arbitrary number of subsamples, in turn each pointing to some piece of store data on disk) are supported.

All data pieces within a column have the same data type. For `bytes` columns, there is no distinction between `'variable_shape'` and `'fixed_shape'` schema types. Values are allowed to take on a value of any size so long as the datatype and contents are valid for the schema definition.

Parameters

- **name** (*str*) – Name assigned to the column
- **contains_subsamples** (*bool, optional*) – True if the column column should store data in a nested structure. In this scheme, a sample key is used to index an arbitrary number of subsamples which map some (sub)key to a piece of data. If False, sample keys map directly to a single piece of data; essentially acting as a single level key/value store. By default, False.
- **backend** (*Optional[str], optional*) – ADVANCED USERS ONLY, backend format code to use for column data. If None, automatically inferred and set based on data shape and type. by default None
- **backend_options** (*Optional[dict], optional*) – ADVANCED USERS ONLY, filter opts to apply to column data. If None, automatically inferred and set based on data shape and type. by default None

Returns instance object of the initialized column.

Return type *Columns*

add_ndarray_column(*name: str, shape: Optional[Union[int, tuple]] = None, dtype: Optional[numpy.dtype] = None, prototype: Optional[numpy.ndarray] = None, variable_shape: bool = False, contains_subsamples: bool = False, *, backend: Optional[str] = None, backend_options: Optional[dict] = None*)

Initializes a `numpy.ndarray` container column.

Columns are created in order to store some arbitrary collection of data pieces. In this case, we store `numpy.ndarray` data. Items need not be related to each-other in any direct capacity; the only criteria hangar requires is that all pieces of data stored in the column have a compatible schema with each-other (more on this below). Each piece of data is indexed by some key (either user defined or automatically generated depending on the user's preferences). Both single level stores (sample keys mapping to data on disk) and nested stores (where some sample key maps to an arbitrary number of subsamples, in turn each pointing to some piece of store data on disk) are supported.

All data pieces within a column have the same data type and number of dimensions. The size of each dimension can be either fixed (the default behavior) or variable per sample. For fixed dimension sizes, all data pieces written to the column must have the same shape & size which was specified at the time the column column was initialized. Alternatively, variable sized columns can write data pieces with dimensions of any size (up to a specified maximum).

Parameters

- **name** (*str*) – The name assigned to this column.
- **shape** (*Optional[Union[int, Tuple[int]]]*) – The shape of the data samples which will be written in this column. This argument and the *dtype* argument are required if a *prototype* is not provided, defaults to None.
- **dtype** (Optional[`numpy.dtype`]) – The datatype of this column. This argument and the *shape* argument are required if a *prototype* is not provided., defaults to None.

- **prototype** (Optional[numpy.ndarray]) – A sample array of correct datatype and shape which will be used to initialize the column storage mechanisms. If this is provided, the *shape* and *dtype* arguments must not be set, defaults to None.
- **variable_shape** (*bool, optional*) – If this is a variable sized column. If true, a the maximum shape is set from the provided `shape` or `prototype` argument. Any sample added to the column can then have dimension sizes <= to this initial specification (so long as they have the same rank as what was specified) defaults to False.
- **contains_subsamples** (*bool, optional*) – True if the column column should store data in a nested structure. In this scheme, a sample key is used to index an arbitrary number of subsamples which map some (sub)key to some piece of data. If False, sample keys map directly to a single piece of data; essentially acting as a single level key/value store. By default, False.
- **backend** (*Optional[str], optional*) – ADVANCED USERS ONLY, backend format code to use for column data. If None, automatically inferred and set based on data shape and type. by default None
- **backend_options** (*Optional[dict], optional*) – ADVANCED USERS ONLY, filter opts to apply to column data. If None, automatically inferred and set based on data shape and type. by default None

Returns instance object of the initialized column.

Return type *Columns*

add_str_column (*name: str, contains_subsamples: bool = False, *, backend: Optional[str] = None, backend_options: Optional[dict] = None*)

Initializes a `str` container column

Columns are created in order to store some arbitrary collection of data pieces. In this case, we store `str` data. Items need not be related to each-other in any direct capacity; the only criteria hangar requires is that all pieces of data stored in the column have a compatible schema with each-other (more on this below). Each piece of data is indexed by some key (either user defined or automatically generated depending on the user's preferences). Both single level stores (sample keys mapping to data on disk) and nested stores (where some sample key maps to an arbitrary number of subsamples, in turn each pointing to some piece of store data on disk) are supported.

All data pieces within a column have the same data type. For `str` columns, there is no distinction between `'variable_shape'` and `'fixed_shape'` schema types. Values are allowed to take on a value of any size so long as the datatype and contents are valid for the schema definition.

Parameters

- **name** (*str*) – Name assigned to the column
- **contains_subsamples** (*bool, optional*) – True if the column column should store data in a nested structure. In this scheme, a sample key is used to index an arbitrary number of subsamples which map some (sub)key to a piece of data. If False, sample keys map directly to a single piece of data; essentially acting as a single level key/value store. By default, False.
- **backend** (*Optional[str], optional*) – ADVANCED USERS ONLY, backend format code to use for column data. If None, automatically inferred and set based on data shape and type. by default None
- **backend_options** (*Optional[dict], optional*) – ADVANCED USERS ONLY, filter opts to apply to column data. If None, automatically inferred and set based on data shape and type. by default None

Returns instance object of the initialized column.

Return type *Columns*

property branch_name

Branch this write enabled checkout's staging area was based on.

Returns name of the branch whose commit `HEAD` changes are staged from.

Return type str

close() → None

Close all handles to the writer checkout and release the writer lock.

Failure to call this method after the writer checkout has been used will result in a lock being placed on the repository which will not allow any writes until it has been manually cleared.

property columns

Provides access to column interaction object.

Can be used to either return the columns accessor for all elements or a single column instance by using dictionary style indexing.

```
>>> co = repo.checkout(write=True)
>>> cols = co.columns
>>> len(cols)
0
>>> fooCol = co.add_ndarray_column('foo', shape=(10, 10), dtype=np.uint8)
>>> len(co.columns)
1
>>> len(co)
1
>>> list(co.columns.keys())
['foo']
>>> list(co.keys())
['foo']
>>> fooCol = co.columns['foo']
>>> fooCol.dtype
np.fooDtype
>>> fooCol = cols.get('foo')
>>> fooCol.dtype
np.fooDtype
>>> 'foo' in co.columns
True
>>> 'bar' in co.columns
False
```

See also:

The class *Columns* contains all methods accessible by this property accessor

Returns the columns object which behaves exactly like a columns accessor class but which can be invalidated when the writer lock is released.

Return type *Columns*

commit(*commit_message: str*) → str

Commit the changes made in the staging area on the checkout branch.

Parameters **commit_message** (*str, optional*) – user proved message for a log of what was changed in this commit. Should a fast forward commit be possible, this will NOT be added to fast-forward `HEAD`.

Returns The commit hash of the new commit.

Return type str

Raises **RuntimeError** – If no changes have been made in the staging area, no commit occurs.

property commit_hash

Commit hash which the staging area of *branch_name* is based on.

Returns commit hash

Return type str

property diff

Access the differ methods which are aware of any staged changes.

See also:

The class *hangar.diff.WriterUserDiff* contains all methods accessible by this property accessor

Returns weakref proxy to the differ object (and contained methods) which behaves exactly like the differ class but which can be invalidated when the writer lock is released.

Return type *WriterUserDiff*

get(*keys*, *default=None*, *except_missing=False*)

View of sample data across columns gracefully handling missing sample keys.

Please see *__getitem__()* for full description. This method is identical with a single exception: if a sample key is not present in an column, this method will plane a null `None` value in it's return slot rather than throwing a `KeyError` like the dict style access does.

Parameters

- **keys** – sequence of column name (and optionally) sample key(s) or sequence of list/tuple of column name, sample keys(s) which should be retrieved in the operation.

  Please see detailed explanation in *__getitem__()* for full explanation of accepted argument format / result types.

- **default** (*Any, optional*) – default value to insert in results for the case where some column name / sample key is not found, and the *except_missing* parameter is set to False.

- **except_missing** (*bool, optional*) – If False, will not throw exceptions on missing sample key value. Will raise KeyError if True and missing key found.

Returns

- *Columns* – single column parameter, no samples specified
- *Any* – Single column specified, single sample key specified
- *List[Any]* – arbitrary columns, multiple samples array data for each sample is returned in same order sample keys are received.

items()

Generator yielding tuple of (name, accessor object) of every column

keys()

Generator yielding the name (key) of every column

log(*branch: str = None*, *commit: str = None*, ***, *return_contents: bool = False*, *show_time: bool = False*, *show_user: bool = False*) → Optional[dict]

Displays a pretty printed commit log graph to the terminal.

Note: For programatic access, the return_contents value can be set to true which will retrieve relevant commit specifications as dictionary elements.

if Neither *branch* nor *commit* arguments are supplied, the branch which is currently checked out for writing will be used as default.

Parameters

- **branch** (*str, optional*) – The name of the branch to start the log process from. (Default value = None)
- **commit** (*str, optional*) – The commit hash to start the log process from. (Default value = None)
- **return_contents** (*bool, optional, kwarg only*) – If true, return the commit graph specifications in a dictionary suitable for programatic access/evaluation.
- **show_time** (*bool, optional, kwarg only*) – If true and return_contents is False, show the time of each commit on the printed log graph
- **show_user** (*bool, optional, kwarg only*) – If true and return_contents is False, show the committer of each commit on the printed log graph

Returns Dict containing the commit ancestor graph, and all specifications.

Return type Optional[dict]

merge (*message: str*, *dev_branch: str*) → str

Merge the currently checked out commit with the provided branch name.

If a fast-forward merge is possible, it will be performed, and the commit message argument to this function will be ignored.

Parameters

- **message** (*str*) – commit message to attach to a three-way merge
- **dev_branch** (*str*) – name of the branch which should be merge into this branch (ie *master*)

Returns commit hash of the new commit for the *master* branch this checkout was started from.

Return type str

reset_staging_area (*, *force=False*) → str

Perform a hard reset of the staging area to the last commit head.

After this operation completes, the writer checkout will automatically close in the typical fashion (any held references to :attr:`column` or :attr:`metadata` objects will finalize and destruct as normal), In order to perform any further operation, a new checkout needs to be opened.

Warning: This operation is IRREVERSIBLE. all records and data which are note stored in a previous commit will be permanently deleted.

Returns Commit hash of the head which the staging area is reset to.

Return type str

Raises `RuntimeError` – If no changes have been made to the staging area, No-Op.

values()

Generator yielding accessor object of every column

Columns

class Columns

Common access patterns and initialization/removal of columns in a checkout.

This object is the entry point to all data stored in their individual columns. Each column contains a common schema which dictates the general shape, dtype, and access patters which the backends optimize access for. The methods contained within allow us to create, remove, query, and access these collections of common data pieces.

__contains__(*key: str*) → bool

Determine if a column with a particular name is stored in the checkout

Parameters **key** (*str*) – name of the column to check for

Returns True if a column with the provided name exists in the checkout, otherwise False.

Return type bool

__delitem__(*key: str*) → str

Remove a column and all data records if write-enabled process.

Parameters **key** (*str*) – Name of the column to remove from the repository. This will remove all records from the staging area (though the actual data and all records are still accessible) if they were previously committed.

Returns If successful, the name of the removed column.

Return type str

Raises **PermissionError** – If any enclosed column is opened in a connection manager.

__getitem__(*key: str*) → Union[NestedSampleWriter, FlatSubsampleWriter, FlatSampleWriter]

Dict style access to return the column object with specified key/name.

Parameters **key** (*string*) – name of the column object to get.

Returns The object which is returned depends on the mode of checkout specified. If the column was checked out with write-enabled, return writer object, otherwise return read only object.

Return type ModifierTypes

__len__() → int

Get the number of column columns contained in the checkout.

property contains_remote_references

Dict of bool indicating data reference locality in each column.

Returns For each column name key, boolean value where False indicates all samples in column exist locally, True if some reference remote sources.

Return type Mapping[str, bool]

delete(*column: str*) → str

Remove the column and all data contained within it.

Parameters **column** (*str*) – name of the column to remove

Returns name of the removed column

Return type str

Raises

- `PermissionError` – If any enclosed column is opened in a connection manager.
- `KeyError` – If a column does not exist with the provided name

get (*name: str*) → Union[NestedSampleWriter, FlatSubsampleWriter, FlatSampleWriter]

Returns a column access object.

This can be used in lieu of the dictionary style access.

Parameters **name** (*str*) – name of the column to return

Returns ColumnData accessor (set to read or write mode as appropriate) which governs interaction with the data

Return type ModifierTypes

property iswriteable

Bool indicating if this column object is write-enabled. Read-only attribute.

items () → Iterable[Tuple[str, Union[NestedSampleWriter, FlatSubsampleWriter, FlatSampleWriter]]]

Generator providing access to column_name, `Columns`

Yields *Iterable[Tuple[str, ModifierTypes]]* – returns two tuple of all all column names/object pairs in the checkout.

keys () → List[str]

list all column keys (names) in the checkout

Returns list of column names

Return type List[str]

property remote_sample_keys

Determine columns samples names which reference remote sources.

Returns dict where keys are column names and values are iterables of samples in the column containing remote references

Return type Mapping[str, Iterable[Union[int, str]]]

values () → Iterable[Union[NestedSampleWriter, FlatSubsampleWriter, FlatSampleWriter]]

Yield all column object instances in the checkout.

Yields *Iterable[ModifierTypes]* – Generator of ColumnData accessor objects (set to read or write mode as appropriate)

Flat Column Layout Container

class FlatSampleWriter

__contains__ (*key: Union[str, int]*) → bool

Determine if a key is a valid sample name in the column.

__delitem__ (*key: Union[str, int]*) → None

Remove a sample from the column. Convenience method to `delete()`.

See also:

`pop()` to return a value and then delete it in the same operation

Parameters **key** (*KeyType*) – Name of the sample to remove from the column.

`__getitem__`(*key: Union[str, int]*)
Retrieve data for some sample key via dict style access conventions.

See also:

`get()`

Parameters **`key`** (*`KeyType`*) – Sample key to retrieve from the column.

Returns Data corresponding to the provided sample key.

Return type value

Raises **`KeyError`** – if no sample with the requested key exists.

`__iter__`() → Iterable[Union[str, int]]
Create iterator yielding an column sample keys.

Yields *Iterable[KeyType]* – Sample key contained in the column.

`__len__`() → int
Check how many samples are present in a given column.

`__setitem__`(*key*, *value*)
Store a piece of data in a column.

See also:

`update()` for an implementation analogous to python's built in `dict.update()` method which accepts a dict or iterable of key/value pairs to add in the same operation.

Parameters

- **`key`** – name to assign to the sample (assuming the column accepts named samples), If str, can only contain alpha-numeric ascii characters (in addition to '-', '.', '_'). Integer key must be >= 0. by default, None
- **`value`** – data to store as a sample in the column.

`append`(*value*) → Union[str, int]
Store some data in a sample with an automatically generated key.

This method should only be used if the context some piece of data is used in is independent from its value (i.e. when reading data back, there is no useful information which needs to be conveyed between the data source's name/id and the value of that piece of information.) Think carefully before going this route, as this posit does not apply to many common use cases.

To store the data with a user defined key, use *`update()`* or *`__setitem__()`*

Parameters **`value`** – Piece of data to store in the column.

Returns Name of the generated key this data is stored with.

Return type KeyType

`property backend`
Code indicating which backing store is used when writing data.

`property backend_options`
Filter / Compression options applied to backend when writing data.

`change_backend`(*backend: str*, *backend_options: Optional[dict] = None*)
Change the default backend and filters applied to future data writes.

> **Warning:** This method is meant for advanced users only. Please refer to the hangar backend codebase for information on accepted parameters and options.

Parameters

- **backend** (*str*) – Backend format code to swtich to.
- **backend_options** (*Optional[dict]*) – Backend option specification to use (if specified). If left to default value of None, then default options for backend are automatically used.

Raises

- **RuntimeError** – If this method was called while this column is invoked in a context manager
- **ValueError** – If the backend format code is not valid.

property column

Name of the column.

property column_layout

Column layout type ('nested', 'flat', etc).

property column_type

Data container type of the column ('ndarray', 'str', etc).

property contains_remote_references

Bool indicating if all samples in column exist on local disk.

The data associated with samples referencing some remote server will need to be downloaded (`fetched` in the hangar vocabulary) before they can be read into memory.

Returns False if at least one sample in the column references data stored on some remote server. True if all sample data is available on the machine's local disk.

Return type bool

property contains_subsamples

Bool indicating if sub-samples are contained in this column container.

property dtype

Dtype of the columns data (np.float, str, etc).

get (*key: Union[str, int], default=None*)

Retrieve the data associated with some sample key

Parameters

- **key** (*KeyType*) – The name of the subsample(s) to retrieve. Passing a single subsample key will return the stored data value.
- **default** (*Any*) – if a *key* parameter is not found, then return this value instead. By default, None.

Returns data data stored under subsample key if key exists, else default value if not found.

Return type value

property iswriteable

Bool indicating if this column object is write-enabled.

items (*local: bool = False*) → Iterable[Tuple[Union[str, int], Any]]
Generator yielding (name, data) tuple for every subsample.

Parameters **local** (*bool, optional*) – If True, returned keys/values will only correspond to data which is available for reading on the local disk, No attempt will be made to read data existing on a remote server, by default False.

Yields *Iterable[Tuple[KeyType, Any]]* – Name and stored value for every subsample inside the sample.

keys (*local: bool = False*) → Iterable[Union[str, int]]
Generator yielding the name (key) of every subsample.

Parameters **local** (*bool, optional*) – If True, returned keys will only correspond to data which is available for reading on the local disk, by default False.

Yields *Iterable[KeyType]* – Keys of one subsample at a time inside the sample.

pop (*key: Union[str, int]*)
Retrieve some value for some key(s) and delete it in the same operation.

Parameters **key** (*KeysType*) – Sample key to remove

Returns Upon success, the value of the removed key.

Return type value

Raises **KeyError** – If there is no sample with some key in the column.

property remote_reference_keys
Compute sample names whose data is stored in a remote server reference.

Returns list of sample keys in the column whose data references indicate they are stored on a remote server.

Return type Tuple[KeyType]

property schema_type
Schema type of the contained data (‘variable_shape’, ‘fixed_shape’, etc).

property shape
(Max) shape of data that can (is) written in the column.

update (*other=None*, ***kwargs*)
Store some data with the key/value pairs from other, overwriting existing keys.

update() implements functionality similar to python’s builtin dict.update() method, accepting either a dictionary or other iterable (of length two) listing key / value pairs.

Parameters

- **other** – Accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). mapping sample names to data value instances instances, If sample name is string type, can only contain alpha-numeric ascii characters (in addition to ‘-‘, ‘.’, ‘_’). Int key must be >= 0. By default, None.
- ****kwargs** – keyword arguments provided will be saved with keywords as sample keys (string type only) and values as np.array instances.

values (*local: bool = False*) → Iterable[Any]
Generator yielding the data for every subsample.

Parameters **local** (*bool, optional*) – If True, returned values will only correspond to data which is available for reading on the local disk. No attempt will be made to read data existing on a remote server, by default False.

Yields *Iterable[Any]* – Values of one subsample at a time inside the sample.

Nested Column Layout Container

class NestedSampleWriter

`__contains__`(*key: Union[str, int]*) → bool
Determine if some sample key exists in the column.

`__delitem__`(*key: Union[str, int]*)
Remove a sample (including all contained subsamples) from the column.

See also:

`pop()` for alternative implementing a simultaneous get value and delete operation.

`__getitem__`(*key: Union[str, int]*) → *hangar.columns.layout_nested.FlatSubsampleReader*
Get the sample access class for some sample key.

Parameters **key** (*KeyType*) – Name of sample to retrieve

Returns Sample accessor corresponding to the given key

Return type *FlatSubsampleReader*

Raises **`KeyError`** – If no sample with the provided key exists.

`__iter__`() → Iterable[Union[str, int]]
Create iterator yielding an column sample keys.

Yields *Iterable[KeyType]* – Sample key contained in the column.

`__len__`() → int
Find number of samples in the column

`__setitem__`(*key, value*) → None
Store some subsample key / subsample data map, overwriting existing keys.

See also:

`update()` for alternative syntax for setting values.

property backend
Code indicating which backing store is used when writing data.

property backend_options
Filter / Compression options applied to backend when writing data.

change_backend(*backend: str, backend_options: Optional[dict] = None*)
Change the default backend and filters applied to future data writes.

> **Warning:** This method is meant for advanced users only. Please refer to the hangar backend codebase for information on accepted parameters and options.

Parameters

- **backend** (*str*) – Backend format code to swtich to.
- **backend_options** – Backend option specification to use (if specified). If left to default value of None, then default options for backend are automatically used.

Raises

- `RuntimeError` – If this method was called while this column is invoked in a context manager
- `ValueError` – If the backend format code is not valid.

property column
Name of the column.

property column_layout
Column layout type ('nested', 'flat', etc).

property column_type
Data container type of the column ('ndarray', 'str', etc).

property contains_remote_references
Bool indicating all subsamples in sample column exist on local disk.

The data associated with subsamples referencing some remote server will need to be downloaded (`fetched` in the hangar vocabulary) before they can be read into memory.

Returns False if at least one subsample in the column references data stored on some remote server. True if all sample data is available on the machine's local disk.

Return type bool

property contains_subsamples
Bool indicating if sub-samples are contained in this column container.

property dtype
Dtype of the columns data (np.float, str, etc).

get(*key: Union[str, int, ellipsis, slice], default: Any = None*) → *hangar.columns.layout_nested.FlatSubsampleReader*
Retrieve data for some sample key(s) in the column.

Parameters

- **key** (*GetKeysType*) – The name of the subsample(s) to retrieve
- **default** (*Any*) – if a *key* parameter is not found, then return this value instead. By default, None.

Returns Sample accessor class given by name `key` which can be used to access subsample data.

Return type *FlatSubsampleReader*

property iswriteable
Bool indicating if this column object is write-enabled.

items(*local: bool = False*) → Iterable[Tuple[Union[str, int], Any]]
Generator yielding (name, data) tuple for every subsample.

Parameters **local** (*bool, optional*) – If True, returned keys/values will only correspond to data which is available for reading on the local disk, No attempt will be made to read data existing on a remote server, by default False.

Yields *Iterable[Tuple[KeyType, Any]]* – Name and stored value for every subsample inside the sample.

keys(*local: bool = False*) → Iterable[Union[str, int]]
Generator yielding the name (key) of every subsample.

Parameters **local** (*bool, optional*) – If True, returned keys will only correspond to data which is available for reading on the local disk, by default False.

Yields *Iterable[KeyType]* – Keys of one subsample at a time inside the sample.

property num_subsamples

Calculate total number of subsamples existing in all samples in column

pop (*key: Union[str, int]*) → Dict[Union[str, int], Any]

Retrieve some value for some key(s) and delete it in the same operation.

Parameters **key** (*KeysType*) – sample key to remove

Returns Upon success, a nested dictionary mapping sample names to a dict of subsample names and subsample values for every sample key passed into this method.

Return type Dict[KeyType, KeyArrMap]

property remote_reference_keys

Compute subsample names whose data is stored in a remote server reference.

Returns list of subsample keys in the column whose data references indicate they are stored on a remote server.

Return type Tuple[KeyType]

property schema_type

Schema type of the contained data (‘variable_shape’, ‘fixed_shape’, etc).

property shape

(Max) shape of data that can (is) written in the column.

update (*other=None*, ***kwargs*) → None

Store some data with the key/value pairs, overwriting existing keys.

`update()` implements functionality similar to python’s builtin `dict.update()` method, accepting either a dictionary or other iterable (of length two) listing key / value pairs.

Parameters

- **other** – Dictionary mapping sample names to subsample data maps. Or Sequence (list or tuple) where element one is the sample name and element two is a subsample data map.
- ****kwargs** – keyword arguments provided will be saved with keywords as sample keys (string type only) and values as a mapping of subarray keys to data values.

values (*local: bool = False*) → Iterable[Any]

Generator yielding the tensor data for every subsample.

Parameters **local** (*bool, optional*) – If True, returned values will only correspond to data which is available for reading on the local disk. No attempt will be made to read data existing on a remote server, by default False.

Yields *Iterable[Any]* – Values of one subsample at a time inside the sample.

class FlatSubsampleWriter

__delitem__ (*key: Union[str, int]*)

Remove a subsample from the column.`.

See also:

`pop()` to simultaneously get a keys value and delete it.

Parameters **key** (*KeyType*) – Name of the sample to remove from the column.

__getitem__(*key: Union[str, int, ellipsis, slice]*) → Union[Any, Dict[Union[str, int], Any]]
Retrieve data for some subsample key via dict style access conventions.

See also:

get()

Parameters **key** (*GetKeysType*) – Sample key to retrieve from the column. Alternatively, `slice` syntax can be used to retrieve a selection of subsample keys/values. An empty slice (`: == slice(None)`) or `Ellipsis (...)` will return all subsample keys/values. Passing a non-empty slice (`[1:5] == slice(1, 5)`) will select keys to retrieve by enumerating all subsamples and retrieving the element (key) for each step across the range. Note: order of enumeration is not guaranteed; do not rely on any ordering observed when using this method.

Returns Sample data corresponding to the provided key. or dictionary of subsample keys/data if Ellipsis or slice passed in as key.

Return type Union[Any, Dict[KeyType, Any]]

Raises **KeyError** – if no sample with the requested key exists.

__setitem__(*key*, *value*)
Store data as a subsample. Convenience method to `add()`.

See also:

update() for an implementation analogous to python's built in `dict.update()` method which accepts a dict or iterable of key/value pairs to add in the same operation.

Parameters

- **key** – Key (name) of the subsample to add to the column.
- **value** – Data to add as the sample.

append(*value*) → Union[str, int]
Store some data in a subsample with an automatically generated key.

This method should only be used if the context some piece of data is used in is independent from its value (ie. when reading data back, there is no useful information which needs to be conveyed between the data source's name/id and the value of that piece of information.) Think carefully before going this route, as this posit does not apply to many common use cases.

See also:

In order to store the data with a user defined key, use *update()* or *__setitem__()*

Parameters **value** – Piece of data to store in the column.

Returns Name of the generated key this data is stored with.

Return type KeyType

property column
Name of the column.

property contains_remote_references

Bool indicating all subsamples in sample column exist on local disk.

The data associated with subsamples referencing some remote server will need to be downloaded (`fetched` in the hangar vocabulary) before they can be read into memory.

Returns False if at least one subsample in the column references data stored on some remote server. True if all sample data is available on the machine's local disk.

Return type bool

property data

Return dict mapping every subsample key / data value stored in the sample.

Returns Dictionary mapping subsample name(s) (keys) to their stored values as `numpy.ndarray` instances.

Return type Dict[KeyType, Any]

get(*key: Union[str, int]*, *default=None*)

Retrieve the data associated with some subsample key

Parameters

- **key** (*GetKeysType*) – The name of the subsample(s) to retrieve. Passing a single subsample key will return the stored `numpy.ndarray`
- **default** – if a *key* parameter is not found, then return this value instead. By default, None.

Returns data stored under subsample key if key exists, else default value if not found.

Return type value

property iswriteable

Bool indicating if this column object is write-enabled.

items(*local: bool = False*) → Iterable[Tuple[Union[str, int], Any]]

Generator yielding (name, data) tuple for every subsample.

Parameters local (*bool, optional*) – If True, returned keys/values will only correspond to data which is available for reading on the local disk, No attempt will be made to read data existing on a remote server, by default False.

Yields *Iterable[Tuple[KeyType, Any]]* – Name and stored value for every subsample inside the sample.

keys(*local: bool = False*) → Iterable[Union[str, int]]

Generator yielding the name (key) of every subsample.

Parameters local (*bool, optional*) – If True, returned keys will only correspond to data which is available for reading on the local disk, by default False.

Yields *Iterable[KeyType]* – Keys of one subsample at a time inside the sample.

pop(*key: Union[str, int]*)

Retrieve some value for some key(s) and delete it in the same operation.

Parameters key (*KeysType*) – Sample key to remove

Returns Upon success, the value of the removed key.

Return type value

property remote_reference_keys

Compute subsample names whose data is stored in a remote server reference.

Returns list of subsample keys in the column whose data references indicate they are stored on a remote server.

Return type Tuple[KeyType]

property sample

Name of the sample this column subsamples are stured under.

update(*other=None*, ***kwargs*)

Store data with the key/value pairs, overwriting existing keys.

`update()` implements functionality similar to python's builtin `dict.update()` method, accepting either a dictionary or other iterable (of length two) listing key / value pairs.

Parameters

- **other** – Accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). mapping sample names to data values, If sample name is string type, can only contain alpha-numeric ascii characters (in addition to '-', '.', '_'). Int key must be >= 0. By default, None.
- ****kwargs** – keyword arguments provided will be saved with keywords as subsample keys (string type only) and values as np.array instances.

values(*local: bool = False*) → Iterable[Any]

Generator yielding the data for every subsample.

Parameters local (`bool, optional`) – If True, returned values will only correspond to data which is available for reading on the local disk. No attempt will be made to read data existing on a remote server, by default False.

Yields *Iterable[Any]* – Values of one subsample at a time inside the sample.

Differ

class WriterUserDiff

Methods diffing contents of a `WriterCheckout` instance.

These provide diffing implementations to compare the current `HEAD` of a checkout to a branch, commit, or the staging area `"base"` contents. The results are generally returned as a nested set of named tuples. In addition, the `status()` method is implemented which can be used to quickly determine if there are any uncommitted changes written in the checkout.

When diffing of commits or branches is performed, if there is not a linear history of commits between current `HEAD` and the diff commit (ie. a history which would permit a `"fast-forward" merge`), the result field named `conflict` will contain information on any merge conflicts that would exist if staging area `HEAD` and the (compared) `"dev" HEAD` were merged "right now". Though this field is present for all diff comparisons, it can only contain non-empty values in the cases where a three way merge would need to be performed.

```
Fast Forward is Possible
========================


    (master)          (foo)
a ----- b ----- c ----- d


3-Way Merge Required
====================

```

(continues on next page)

(continued from previous page)

```
                    (master)
a ----- b ----- c ----- d
        \
         \               (foo)
          \----- ee ----- ff
```

branch(*dev_branch: str*) → hangar.diff.DiffAndConflicts

Compute diff between HEAD and branch, returning user-facing results.

Parameters **dev_branch** (*str*) – name of the branch whose HEAD will be used to calculate the diff of.

Returns two-tuple of `diff`, `conflict` (if any) calculated in the diff algorithm.

Return type DiffAndConflicts

Raises **ValueError** – If the specified `dev_branch` does not exist.

commit(*dev_commit_hash: str*) → hangar.diff.DiffAndConflicts

Compute diff between HEAD and commit, returning user-facing results.

Parameters **dev_commit_hash** (*str*) – hash of the commit to be used as the comparison.

Returns two-tuple of `diff`, `conflict` (if any) calculated in the diff algorithm.

Return type DiffAndConflicts

Raises **ValueError** – if the specified `dev_commit_hash` is not a valid commit reference.

staged() → hangar.diff.DiffAndConflicts

Return diff of staging area to base, returning user-facing results.

Returns two-tuple of `diff`, `conflict` (if any) calculated in the diff algorithm.

Return type DiffAndConflicts

status() → str

Determine if changes have been made in the staging area

If the contents of the staging area and it's parent commit are the same, the status is said to be "CLEAN". If even one column or metadata record has changed however, the status is "DIRTY".

Returns "CLEAN" if no changes have been made, otherwise "DIRTY"

Return type str

Bulk Importer

Bulk importer methods to ingest large quantities of data into Hangar.

The following module is designed to address challenges inherent to writing massive amounts of data to a hangar repository via the standard API. Since write-enabled checkouts are limited to processing in a single thread, the time required to import hundreds of Gigabytes (or Terabytes) of data into Hangar (from external sources) can become prohibitivly long. This module implements a multi-processed importer which reduces import time nearly linearly with the number of CPU cores allocated on a machine.

There are a number of challenges to overcome:

1. How to validate data against a column schema?
   - Does the column exist?
   - Are the key(s) valid?

    - Is the data a valid type/shape/precision valid for the the selected column schema?

2. How to handle duplicated data?
    - If an identical piece of data is recorded in the repository already, only record the sample reference (do not write the data to disk again).
    - If the bulk import method would write identical pieces of data to the repository multiple times, and the data does not already exist, then that piece of content should only be written to disk once. Only sample references should be saved after that.
3. How to handle transactionality?
    - What happens if some column, sample keys, or data piece is invalid and cannot be written as desired?
    - How to rollback partial changes if the process is inturupted in the middle of a bulk import operation?
4. How to limit memory usage if many processes are trying to load and write large tensors?

Rough outline of steps:

1. Validate UDF & Argument Signature
2. Read, Validate, and Hash UDF results –> Task Recipe
3. Prune Recipe
4. Read, Validate, Write Data to Isolated Backend Storage
5. Record Sample References in Isolated Environment
6. If all successful, make isolated data known to repository core, otherwise abort to starting state.

class UDF_Return(*column: str, key: Union[str, int, Tuple[Union[str, int], Union[str, int]]], data: Union[numpy.ndarray, str, bytes]*)

User-Defined Function return container for bulk importer read functions

Variables

- `column` (`str`) – column name to place data into
- `key` (`Union[KeyType, Tuple[KeyType, KeyType]]`) – key to place flat sample into, or 2-tuple of keys for nested samples
- `data` (`Union[np.ndarray, str, bytes]`) – piece of data to place in the column with the provided key.

property column

Alias for field number 0

property data

Alias for field number 2

property key

Alias for field number 1

run_bulk_import(*repo: Repository, branch_name: str, column_names: List[str], udf: Callable[[. . .], Iterator[UDF_Return]], udf_kwargs: List[dict], *, ncpus: int = 0, autocommit: bool = True*)

Perform a bulk import operation from a given user-defined function.

In order to provide for arbitrary input data sources along with ensuring the core promises of hangar hold we require the following from users:

Define some arbitrary function (ie “user-defined function” / “UDF”) which accepts some arguments and yields data. The UDF must be a generator function, yielding only values which are of `UDF_Return` type. The results yielded by the UDF must be deterministic for a given set of inputs. This includes all values of the *UDF_Return* (`columns` and `keys`, as well as `data`).

A list of input arguments to the UDF must be provided, this is formatted as a sequence (list / tuple) of keyword-arg dictionaries, each of which must be valid when unpacked and bound to the UDF signature. Additionally, all columns must be specified up front. If any columns are named a *UDF_Return* which were not pre-specified, the entire operation will fail.

Notes

- This is an all-or-nothing operation, either all data is successfully read, validated, and written to the storage backends, or none of it is. A single maleformed key or data type/shape will cause the entire import operation to abort.
- The input kwargs should be fairly small (of no consequence to load into memory), data out should be large. The results of the UDF will only be stored in memory for a very short period (just the time it takes to be validated against the column schema and compressed / flushed to disk).
- Every step of the process is executed as a generator, lazily loading data the entire way. If possible, we recomend writing the UDF such that data is not allocated in memory before it is ready to be yielded.
- If it is possible, the task recipe will be pruned and optimized in such a way that iteration over the UDF will be short circuted during the second pass (writing data to the backend). As this can greatly reduce processing time, we recomend trying to yield data pieces which are likely to be unique first from the UDF.

Warning:

- Please be aware that these methods should not be executed within a Jupyter Notebook / Jupyter Lab when running the bulk importer at scale. The internal implemenation makes significant use of multiprocess Queues for work distribution and recording. The heavy loads placed on the system have been observed to place strain on Jupyters ZeroMQ implementation, resulting in random failures which may or may not even display a traceback to indicate failure mode.

  A small sample set of data can be used within jupyter to test an implementation without problems, but for full scale operations it is best run in a script with the operations protected by a `__main__` block.

Examples

```
>>> import os
>>> import numpy as np
>>> from PIL import Image
>>> from hangar.bulk_importer import UDF_Return
```

```
>>> def image_loader(file_path):
...     im = Image.open(file_name)
...     arr = np.array(im.resize(512, 512))
...     im_record = UDF_Return(column='image', key=(category, sample), data=arr)
...     yield im_record
...
...     root, sample_file = os.path.split(file_path)
...     category = os.path.dirname(root)
...     sample_name, _ = os.path.splitext(sample_file)
...     path_record = UDF_Return(column='file_str', key=(category, sample_name),␣
↪data=file_path)
```

(continues on next page)

(continued from previous page)

```
...     yield path_record
...
>>> udf_kwargs = [
...     {'file_path': '/foo/cat/image_001.jpeg'},
...     {'file_path': '/foo/cat/image_002.jpeg'},
...     {'file_path': '/foo/dog/image_001.jpeg'},
...     {'file_path': '/foo/bird/image_011.jpeg'},
...     {'file_path': '/foo/bird/image_003.jpeg'}
... ]
>>> repo = Repository('foo/path/to/repo')
>>> from hangar.bulk_importer import run_bulk_import
>>> run_bulk_import(
...     repo, branch_name='master', column_names=['file_str', 'image'],
...     udf=image_loader, udf_kwargs=udf_kwargs)
```

However, the following will not work, since the output is non-deterministic.

```
>>> def nondeterminstic(x, y):
...     first = str(x * y)
...     yield UDF_Return(column='valstr', key=f'{x}_{y}', data=first)
...
...     second = str(x * y * random())
...     yield UDF_Return(column='valstr', key=f'{x}_{y}', data=second)
...
>>> udf_kwargs = [
...     {'x': 1, 'y': 2},
...     {'x': 1, 'y': 3},
...     {'x': 2, 'y': 4},
... ]
>>> run_bulk_import(
...     repo, branch_name='master', column_names=['valstr'],
...     udf=image_loader, udf_kwargs=udf_kwargs)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: contents returned in subbsequent calls to UDF with identical
  kwargs yielded different results. UDFs MUST generate deterministic
  results for the given inputs. Input kwargs generating this result:
  {'x': 1, 'y': 2}.
```

Not all columns must be returned from every input to the UDF, the number of data pieces yielded can also vary arbitrarily (so long as the results are deterministic for a particular set of inputs)

```
>>> def maybe_load(x_arr, y_arr, sample_name, columns=['default']):
...     for column in columns:
...         arr = np.multiply(x_arr, y_arr)
...         yield UDF_Return(column=column, key=sample_name, data=arr)
...     #
...     # do some strange processing which only outputs another column sometimes
...     if len(columns) == 1:
...         other = np.array(x_arr.shape) * np.array(y_arr.shape)
...         yield UDF_Return(column='strange_column', key=sample_name, data=other)
...
>>> udf_kwargs = [
...     {'x_arr': np.arange(10), 'y_arr': np.arange(10) + 1, 'sample_name':
↪'sample_1'},
...     {'x_arr': np.arange(10), 'y_arr': np.arange(10) + 1, 'sample_name':
↪'sample_2', 'columns': ['foo', 'bar', 'default']},
```

(continues on next page)

(continued from previous page)

```
...     {'x_arr': np.arange(10) * 2, 'y_arr': np.arange(10), 'sample_name':
↪'sample_3'},
... ]
>>> run_bulk_import(
...     repo, branch_name='master',
...     column_names=['default', 'foo', 'bar', 'strange_column'],
...     udf=maybe_load, udf_kwargs=udf_kwargs)
```

Parameters

- **repo** (*'Repository'*) – Initialized repository object to import data into.
- **branch_name** (*str*) – Name of the branch to checkout and import data into.
- **column_names** (*List[str]*) – Names of all columns which data should be saved to.
- **udf** (*UDF_T*) – User-Defined Function (generator style; yielding an arbitrary number of values when iterated on) which is passed an unpacked kwarg dict as input and yields a single *UDF_Return* instance at a time when iterated over. Cannot contain
- **udf_kwargs** (*List[dict]*) – A sequence of keyword argument dictionaries which are individually unpacked as inputs into the user-defined function (UDF). the keyword argument dictionaries
- **ncpus** (*int, optional, default=0*) – Number of Parallel processes to read data files & write to hangar backend stores in. If <= 0, then the default is set to `num_cpus / 2`. The value of this parameter should never exceed the total CPU count of the system. Import time scales mostly linearly with ncpus. Optimal performance is achieved by balancing memory usage of the `UDF` function and backend storage writer processes against the total system memory. generally increase linearly up to
- **autocommit** (*bool, optional, default=True*) – Control whether a commit should be made after successfully importing the specified data to the staging area of the branch.

4.5.4 Read Only Checkout

Checkout

class ReaderCheckout

Checkout the repository as it exists at a particular branch.

This class is instantiated automatically from a repository checkout operation. This object will govern all access to data and interaction methods the user requests.

```
>>> co = repo.checkout()
>>> isinstance(co, ReaderCheckout)
True
```

If a commit hash is provided, it will take precedent over the branch name parameter. If neither a branch not commit is specified, the staging environment's base branch `HEAD` commit hash will be read.

```
>>> co = repo.checkout(commit='foocommit')
>>> co.commit_hash
'foocommit'
>>> co.close()
```

(continues on next page)

(continued from previous page)

```
>>> co = repo.checkout(branch='testbranch')
>>> co.commit_hash
'someothercommithashhere'
>>> co.close()
```

Unlike `WriterCheckout`, any number of `ReaderCheckout` objects can exist on the repository independently. Like the `write-enabled` variant, the `close()` method should be called after performing the necessary operations on the repo. However, as there is no concept of a `lock` for `read-only` checkouts, this is just to free up memory resources, rather than changing recorded access state.

In order to reduce the chance that the python interpreter is shut down without calling `close()`, - a common mistake during ipython / jupyter sessions - an atexit hook is registered to `close()`. If properly closed by the user, the hook is unregistered after completion with no ill effects. So long as a the process is NOT terminated via non-python `SIGKILL`, fatal internal python error, or or special `os exit` methods, cleanup will occur on interpreter shutdown and resources will be freed. If a non-handled termination method does occur, the implications of holding resources varies on a per-OS basis. While no risk to data integrity is observed, repeated misuse may require a system reboot in order to achieve expected performance characteristics.

__contains__(*key*)

Determine if some column name (key) exists in the checkout.

__getitem__(*index*)

Dictionary style access to columns and samples

Checkout object can be thought of as a “dataset” (“dset”) mapping a view of samples across columns.

```
>>> dset = repo.checkout(branch='master')
>>>
# Get an column contained in the checkout.
>>> dset['foo']
ColumnDataReader
>>>
# Get a specific sample from ``'foo'`` (returns a single array)
>>> dset['foo', '1']
np.array([1])
>>>
# Get multiple samples from ``'foo'`` (returns a list of arrays, in order
# of input keys)
>>> dset[['foo', '1'], ['foo', '2'],  ['foo', '324']]
[np.array([1]), np.ndarray([2]), np.ndarray([324])]
>>>
# Get sample from multiple columns, column/data returned is ordered
# in same manner as input of func.
>>> dset[['foo', '1'], ['bar', '1'],  ['baz', '1']]
[np.array([1]), np.ndarray([1, 1]), np.ndarray([1, 1, 1])]
>>>
# Get multiple samples from multiple columns             >>> keys = [(col,␣
↪str(samp)) for samp in range(2) for col in ['foo', 'bar']]
>>> keys
[('foo', '0'), ('bar', '0'), ('foo', '1'), ('bar', '1')]
>>> dset[keys]
[np.array([1]), np.array([1, 1]), np.array([2]), np.array([2, 2])]
```

Arbitrary column layouts are supported by simply adding additional members to the keys for each piece of data. For example, getting data from a column with a nested layout:

```
>>> dset['nested_col', 'sample_1', 'subsample_0']
np.array([1, 0])
>>>
# a sample accessor object can be retrieved at will...
>>> dset['nested_col', 'sample_1']
<class 'FlatSubsampleReader'>(column_name='nested_col', sample_name='sample_1
↪')
>>>
# to get all subsamples in a nested sample use the Ellipsis operator
>>> dset['nested_col', 'sample_1', ...]
{'subsample_0': np.array([1, 0]),
 'subsample_1': np.array([1, 1]),
 ...
 'subsample_n': np.array([1, 255])}
```

Retrieval of data from different column types can be mixed and combined as desired. For example, retrieving data from both flat and nested columns simultaneously:

```
>>> dset[('nested_col', 'sample_1', '0'), ('foo', '0')]
[np.array([1, 0]), np.array([0])]
>>> dset[('nested_col', 'sample_1', ...), ('foo', '0')]
[{'subsample_0': np.array([1, 0]), 'subsample_1': np.array([1, 1])},
 np.array([0])]
>>> dset[('foo', '0'), ('nested_col', 'sample_1')]
[np.array([0]),
 <class 'FlatSubsampleReader'>(column_name='nested_col', sample_name='sample_1
↪')]
```

If a column or data key does not exist, then this method will raise a KeyError. As an alternative, missing keys can be gracefully handeled by calling `get()` instead. This method does not (by default) raise an error if a key is missing. Instead, a (configurable) default value is simply inserted in it's place.

```
>>> dset['foo', 'DOES_NOT_EXIST']
-------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-40-731e6ea62fb8> in <module>
----> 1 res = co['foo', 'DOES_NOT_EXIST']
KeyError: 'DOES_NOT_EXIST'
```

Parameters **index** – column name, sample key(s) or sequence of list/tuple of column name, sample keys(s) which should be retrieved in the operation.

Please see detailed explanation above for full explanation of accepted argument format / result types.

Returns

- `Columns` – single column parameter, no samples specified
- *Any* – Single column specified, single sample key specified
- *List[Any]* – arbitrary columns, multiple samples array data for each sample is returned in same order sample keys are received.

__iter__()

Iterate over column keys

__len__()

Returns number of columns in the checkout.

close() → None

Gracefully close the reader checkout object.

Though not strictly required for reader checkouts (as opposed to writers), closing the checkout after reading will free file handles and system resources, which may improve performance for repositories with multiple simultaneous read checkouts.

property columns

Provides access to column interaction object.

Can be used to either return the columns accessor for all elements or a single column instance by using dictionary style indexing.

```
>>> co = repo.checkout(write=False)
>>> len(co.columns)
1
>>> print(co.columns.keys())
['foo']
>>> fooCol = co.columns['foo']
>>> fooCol.dtype
np.fooDtype
>>> cols = co.columns
>>> fooCol = cols['foo']
>>> fooCol.dtype
np.fooDtype
>>> fooCol = cols.get('foo')
>>> fooCol.dtype
np.fooDtype
```

See also:

The class `Columns` contains all methods accessible by this property accessor

Returns the columns object which behaves exactly like a columns accessor class but which can be invalidated when the writer lock is released.

Return type `Columns`

property commit_hash

Commit hash this read-only checkout's data is read from.

```
>>> co = repo.checkout()
>>> co.commit_hash
foohashdigesthere
```

Returns commit hash of the checkout

Return type str

property diff

Access the differ methods for a read-only checkout.

See also:

The class `ReaderUserDiff` contains all methods accessible by this property accessor

Returns weakref proxy to the differ object (and contained methods) which behaves exactly like the differ class but which can be invalidated when the writer lock is released.

Return type *ReaderUserDiff*

get (*keys*, *default=None*, *except_missing=False*)

View of sample data across columns gracefully handling missing sample keys.

Please see `__getitem__()` for full description. This method is identical with a single exception: if a sample key is not present in an column, this method will plane a null `None` value in it's return slot rather than throwing a `KeyError` like the dict style access does.

Parameters

- **keys** – sequence of column name (and optionally) sample key(s) or sequence of list/tuple of column name, sample keys(s) which should be retrieved in the operation.

  Please see detailed explanation in `__getitem__()` for full explanation of accepted argument format / result types.

- **default** (*Any, optional*) – default value to insert in results for the case where some column name / sample key is not found, and the *except_missing* parameter is set to False.
- **except_missing** (*bool, optional*) – If False, will not throw exceptions on missing sample key value. Will raise KeyError if True and missing key found.

Returns

- *Columns* – single column parameter, no samples specified
- *Any* – Single column specified, single sample key specified
- *List[Any]* – arbitrary columns, multiple samples array data for each sample is returned in same order sample keys are received.

items ()

Generator yielding tuple of (name, accessor object) of every column

keys ()

Generator yielding the name (key) of every column

log (*branch: str = None*, *commit: str = None*, ***, *return_contents: bool = False*, *show_time: bool = False*, *show_user: bool = False*) → Optional[dict]

Displays a pretty printed commit log graph to the terminal.

Note: For programatic access, the return_contents value can be set to true which will retrieve relevant commit specifications as dictionary elements.

if Neither *branch* nor *commit* arguments are supplied, the commit digest of the currently reader checkout will be used as default.

Parameters

- **branch** (*str, optional*) – The name of the branch to start the log process from. (Default value = None)
- **commit** (*str, optional*) – The commit hash to start the log process from. (Default value = None)
- **return_contents** (*bool, optional, kwarg only*) – If true, return the commit graph specifications in a dictionary suitable for programatic access/evaluation.
- **show_time** (*bool, optional, kwarg only*) – If true and return_contents is False, show the time of each commit on the printed log graph
- **show_user** (*bool, optional, kwarg only*) – If true and return_contents is False, show the committer of each commit on the printed log graph

Returns Dict containing the commit ancestor graph, and all specifications.

Return type Optional[dict]

values()
Generator yielding accessor object of every column

Flat Column Layout Container

class FlatSampleReader
Class implementing get access to data in a column.

This class exposes the standard API to access data stored in a single level key / value mapping column. Usage is modeled after the python `dict` style syntax – with a few additional utility and inspection methods and properties added. Methods named after those of a python `dict` have syntactically identical arguments and behavior to that of the standard library.

If not opened in a `write-enabled` checkout, then attempts to add or delete data or container properties will raise an exception (in the form of a `PermissionError`). No changes will be propogated unless a `write-enabled` checkout is used.

This object can be serialized – pickled – for parallel processing / reading if opened in a `read-only` checkout. Parallel operations are both thread and process safe, though performance may significantly differ between multithreaded vs multiprocessed code (depending on the backend data is stored in). Attempts to serialize objects opened in `write-enabled` checkouts are not supported and will raise a `PermissionError` if attempted. This behavior is enforced in order to ensure data and record integrity while writing to the repository.

__contains__(*key: Union[str, int]*) → bool
Determine if a key is a valid sample name in the column.

__getitem__(*key: Union[str, int]*)
Retrieve data for some sample key via dict style access conventions.

See also:

get()

Parameters key (*KeyType*) – Sample key to retrieve from the column.

Returns Data corresponding to the provided sample key.

Return type value

Raises **KeyError** – if no sample with the requested key exists.

__iter__() → Iterable[Union[str, int]]
Create iterator yielding an column sample keys.

Yields *Iterable[KeyType]* – Sample key contained in the column.

__len__() → int
Check how many samples are present in a given column.

property backend
Code indicating which backing store is used when writing data.

property backend_options
Filter / Compression options applied to backend when writing data.

property column
Name of the column.

property column_layout
Column layout type ('nested', 'flat', etc).

property column_type
Data container type of the column ('ndarray', 'str', etc).

property contains_remote_references
Bool indicating if all samples in column exist on local disk.

The data associated with samples referencing some remote server will need to be downloaded (`fetched` in the hangar vocabulary) before they can be read into memory.

Returns False if at least one sample in the column references data stored on some remote server. True if all sample data is available on the machine's local disk.

Return type bool

property contains_subsamples
Bool indicating if sub-samples are contained in this column container.

property dtype
Dtype of the columns data (np.float, str, etc).

get(*key: Union[str, int]*, *default=None*)
Retrieve the data associated with some sample key

Parameters

- **key** (*KeyType*) – The name of the subsample(s) to retrieve. Passing a single subsample key will return the stored data value.
- **default** (*Any*) – if a *key* parameter is not found, then return this value instead. By default, None.

Returns data data stored under subsample key if key exists, else default value if not found.

Return type value

property iswriteable
Bool indicating if this column object is write-enabled.

items(*local: bool = False*) → Iterable[Tuple[Union[str, int], Any]]
Generator yielding (name, data) tuple for every subsample.

Parameters **local** (*bool, optional*) – If True, returned keys/values will only correspond to data which is available for reading on the local disk, No attempt will be made to read data existing on a remote server, by default False.

Yields *Iterable[Tuple[KeyType, Any]]* – Name and stored value for every subsample inside the sample.

keys(*local: bool = False*) → Iterable[Union[str, int]]
Generator yielding the name (key) of every subsample.

Parameters **local** (*bool, optional*) – If True, returned keys will only correspond to data which is available for reading on the local disk, by default False.

Yields *Iterable[KeyType]* – Keys of one subsample at a time inside the sample.

property remote_reference_keys
Compute sample names whose data is stored in a remote server reference.

Returns list of sample keys in the column whose data references indicate they are stored on a remote server.

Return type Tuple[KeyType]

`property schema_type`
Schema type of the contained data ('variable_shape', 'fixed_shape', etc).

`property shape`
(Max) shape of data that can (is) written in the column.

`values`(*local: bool = False*) → Iterable[Any]
Generator yielding the data for every subsample.

Parameters **`local`** (`bool, optional`) – If True, returned values will only correspond to data which is available for reading on the local disk. No attempt will be made to read data existing on a remote server, by default False.

Yields *Iterable[Any]* – Values of one subsample at a time inside the sample.

Nested Column Layout Container

`class NestedSampleReader`

`__contains__`(*key: Union[str, int]*) → bool
Determine if some sample key exists in the column.

`__getitem__`(*key: Union[str, int]*) → *hangar.columns.layout_nested.FlatSubsampleReader*
Get the sample access class for some sample key.

Parameters **`key`** (`KeyType`) – Name of sample to retrieve

Returns Sample accessor corresponding to the given key

Return type *FlatSubsampleReader*

Raises **`KeyError`** – If no sample with the provided key exists.

`__iter__`() → Iterable[Union[str, int]]
Create iterator yielding an column sample keys.

Yields *Iterable[KeyType]* – Sample key contained in the column.

`__len__`() → int
Find number of samples in the column

`property backend`
Code indicating which backing store is used when writing data.

`property backend_options`
Filter / Compression options applied to backend when writing data.

`property column`
Name of the column.

`property column_layout`
Column layout type ('nested', 'flat', etc).

`property column_type`
Data container type of the column ('ndarray', 'str', etc).

`property contains_remote_references`
Bool indicating all subsamples in sample column exist on local disk.

The data associated with subsamples referencing some remote server will need to be downloaded (`fetched` in the hangar vocabulary) before they can be read into memory.

Returns False if at least one subsample in the column references data stored on some remote server. True if all sample data is available on the machine's local disk.

Return type bool

property contains_subsamples

Bool indicating if sub-samples are contained in this column container.

property dtype

Dtype of the columns data (np.float, str, etc).

get (*key: Union[str, int, ellipsis, slice], default: Any = None*) → *hangar.columns.layout_nested.FlatSubsampleReader*

Retrieve data for some sample key(s) in the column.

Parameters

- **key** (*GetKeysType*) – The name of the subsample(s) to retrieve
- **default** (*Any*) – if a *key* parameter is not found, then return this value instead. By default, None.

Returns Sample accessor class given by name `key` which can be used to access subsample data.

Return type *FlatSubsampleReader*

property iswriteable

Bool indicating if this column object is write-enabled.

items (*local: bool = False*) → Iterable[Tuple[Union[str, int], Any]]

Generator yielding (name, data) tuple for every subsample.

Parameters local (*bool, optional*) – If True, returned keys/values will only correspond to data which is available for reading on the local disk, No attempt will be made to read data existing on a remote server, by default False.

Yields *Iterable[Tuple[KeyType, Any]]* – Name and stored value for every subsample inside the sample.

keys (*local: bool = False*) → Iterable[Union[str, int]]

Generator yielding the name (key) of every subsample.

Parameters local (*bool, optional*) – If True, returned keys will only correspond to data which is available for reading on the local disk, by default False.

Yields *Iterable[KeyType]* – Keys of one subsample at a time inside the sample.

property num_subsamples

Calculate total number of subsamples existing in all samples in column

property remote_reference_keys

Compute subsample names whose data is stored in a remote server reference.

Returns list of subsample keys in the column whose data references indicate they are stored on a remote server.

Return type Tuple[KeyType]

property schema_type

Schema type of the contained data ('variable_shape', 'fixed_shape', etc).

property shape

(Max) shape of data that can (is) written in the column.

values (*local: bool = False*) → Iterable[Any]

Generator yielding the tensor data for every subsample.

Parameters **local** (*bool, optional*) – If True, returned values will only correspond to data which is available for reading on the local disk. No attempt will be made to read data existing on a remote server, by default False.

Yields *Iterable[Any]* – Values of one subsample at a time inside the sample.

class FlatSubsampleReader

__getitem__(*key: Union[str, int, ellipsis, slice]*) → Union[Any, Dict[Union[str, int], Any]]

Retrieve data for some subsample key via dict style access conventions.

See also:

`get()`

Parameters **key** (*GetKeysType*) – Sample key to retrieve from the column. Alternatively, `slice` syntax can be used to retrieve a selection of subsample keys/values. An empty slice (`: == slice(None)`) or `Ellipsis (...)` will return all subsample keys/values. Passing a non-empty slice (`[1:5] == slice(1, 5)`) will select keys to retrieve by enumerating all subsamples and retrieving the element (key) for each step across the range. Note: order of enumeration is not guaranteed; do not rely on any ordering observed when using this method.

Returns Sample data corresponding to the provided key. or dictionary of subsample keys/data if Ellipsis or slice passed in as key.

Return type Union[Any, Dict[KeyType, Any]]

Raises **KeyError** – if no sample with the requested key exists.

property column

Name of the column.

property contains_remote_references

Bool indicating all subsamples in sample column exist on local disk.

The data associated with subsamples referencing some remote server will need to be downloaded (`fetched` in the hangar vocabulary) before they can be read into memory.

Returns False if at least one subsample in the column references data stored on some remote server. True if all sample data is available on the machine's local disk.

Return type bool

property data

Return dict mapping every subsample key / data value stored in the sample.

Returns Dictionary mapping subsample name(s) (keys) to their stored values as `numpy.ndarray` instances.

Return type Dict[KeyType, Any]

get(*key: Union[str, int], default=None*)

Retrieve the data associated with some subsample key

Parameters

- **key** (*GetKeysType*) – The name of the subsample(s) to retrieve. Passing a single subsample key will return the stored `numpy.ndarray`
- **default** – if a *key* parameter is not found, then return this value instead. By default, None.

Returns data stored under subsample key if key exists, else default value if not found.

Return type value

property iswriteable

Bool indicating if this column object is write-enabled.

items(*local: bool = False*) → Iterable[Tuple[Union[str, int], Any]]

Generator yielding (name, data) tuple for every subsample.

Parameters **local** (*bool, optional*) – If True, returned keys/values will only correspond to data which is available for reading on the local disk, No attempt will be made to read data existing on a remote server, by default False.

Yields *Iterable[Tuple[KeyType, Any]]* – Name and stored value for every subsample inside the sample.

keys(*local: bool = False*) → Iterable[Union[str, int]]

Generator yielding the name (key) of every subsample.

Parameters **local** (*bool, optional*) – If True, returned keys will only correspond to data which is available for reading on the local disk, by default False.

Yields *Iterable[KeyType]* – Keys of one subsample at a time inside the sample.

property remote_reference_keys

Compute subsample names whose data is stored in a remote server reference.

Returns list of subsample keys in the column whose data references indicate they are stored on a remote server.

Return type Tuple[KeyType]

property sample

Name of the sample this column subsamples are stured under.

values(*local: bool = False*) → Iterable[Any]

Generator yielding the data for every subsample.

Parameters **local** (*bool, optional*) – If True, returned values will only correspond to data which is available for reading on the local disk. No attempt will be made to read data existing on a remote server, by default False.

Yields *Iterable[Any]* – Values of one subsample at a time inside the sample.

Differ

class ReaderUserDiff

Methods diffing contents of a `ReaderCheckout` instance.

These provide diffing implementations to compare the current checkout `HEAD` of a to a branch or commit. The results are generally returned as a nested set of named tuples.

When diffing of commits or branches is performed, if there is not a linear history of commits between current `HEAD` and the diff commit (ie. a history which would permit a `"fast-forward" merge`), the result field named `conflict` will contain information on any merge conflicts that would exist if staging area `HEAD` and the (compared) `"dev" HEAD` were merged “right now”. Though this field is present for all diff comparisons, it can only contain non-empty values in the cases where a three way merge would need to be performed.

```
Fast Forward is Possible
========================

    (master)           (foo)
a ----- b ----- c ----- d


3-Way Merge Required
====================

                     (master)
a ----- b ----- c ----- d
        \
         \               (foo)
          \----- ee ----- ff
```

branch (*dev_branch: str*) → hangar.diff.DiffAndConflicts

Compute diff between HEAD and branch name, returning user-facing results.

Parameters **dev_branch** (`str`) – name of the branch whose HEAD will be used to calculate the diff of.

Returns two-tuple of `diff`, `conflict` (if any) calculated in the diff algorithm.

Return type DiffAndConflicts

Raises **ValueError** – If the specified *dev_branch* does not exist.

commit (*dev_commit_hash: str*) → hangar.diff.DiffAndConflicts

Compute diff between HEAD and commit hash, returning user-facing results.

Parameters **dev_commit_hash** (`str`) – hash of the commit to be used as the comparison.

Returns two-tuple of `diff`, `conflict` (if any) calculated in the diff algorithm.

Return type DiffAndConflicts

Raises **ValueError** – if the specified `dev_commit_hash` is not a valid commit reference.

4.5.5 ML Framework Dataloaders

Tensorflow

make_tf_dataset (*columns*, *keys: Sequence[str] = None*, *index_range: slice = None*, *shuffle: bool = True*)

Uses the hangar columns to make a tensorflow dataset. It uses *from_generator* function from *tensorflow.data.Dataset* with a generator function that wraps all the hangar columns. In such instances tensorflow Dataset does shuffle by loading the subset of data which can fit into the memory and shuffle that subset. Since it is not really a global shuffle *make_tf_dataset* accepts a *shuffle* argument which will be used by the generator to shuffle each time it is being called.

> **Warning:** *tf.data.Dataset.from_generator* currently uses *tf.compat.v1.py_func()* internally. Hence the serialization function (*yield_data*) will not be serialized in a GraphDef. Therefore, you won't be able to serialize your model and restore it in a different environment if you use *make_tf_dataset*. The operation must run in the same address space as the Python program that calls tf.compat.v1.py_func(). If you are using distributed TensorFlow, you must run a tf.distribute.Server in the same process as the program that calls tf.compat.v1.py_func() and you must pin the created operation to a device in that server (e.g. using with tf.device():)

Parameters

- **columns** (*Columns* or Sequence) – A column object, a tuple of column object or a list of column objects`
- **keys** (*Sequence[str]*) – An iterable of sample names. If given only those samples will fetched from the column
- **index_range** (*slice*) – A python slice object which will be used to find the subset of column. Argument *keys* takes priority over *index_range* i.e. if both are given, keys will be used and *index_range* will be ignored
- **shuffle** (*bool*) – generator uses this to decide a global shuffle accross all the samples is required or not. But user doesn't have any restriction on doing`column.shuffle()` on the returned column

Examples

```
>>> from hangar import Repository
>>> from hangar import make_tf_dataset
>>> import tensorflow as tf
>>> tf.compat.v1.enable_eager_execution()
>>> repo = Repository('.')
>>> co = repo.checkout()
>>> data = co.columns['mnist_data']
>>> target = co.columns['mnist_target']
>>> tf_dset = make_tf_dataset([data, target])
>>> tf_dset = tf_dset.batch(512)
>>> for bdata, btarget in tf_dset:
...     print(bdata.shape, btarget.shape)
```

Returns

Return type `tf.data.Dataset`

Pytorch

make_torch_dataset (*columns, keys: Sequence[str] = None, index_range: slice = None, field_names: Sequence[str] = None*)

Returns a `torch.utils.data.Dataset` object which can be loaded into a `torch.utils.data.DataLoader`.

Warning: On Windows systems, setting the parameter `num_workers` in the resulting `torch.utils.data.DataLoader` method will result in a RuntimeError or deadlock. This is due to limitations of multi-process start methods on Windows itself. Using the default argument value (`num_workers=0`) will let the DataLoader work in single process mode as expected.

Parameters

- **columns** (*Columns* or Sequence) – A column object, a tuple of column object or a list of column objects.
- **keys** (*Sequence[str]*) – An iterable collection of sample names. If given only those samples will fetched from the column

- **index_range** (*slice*) – A python slice object which will be used to find the subset of column. Argument *keys* takes priority over *range* i.e. if both are given, keys will be used and *range* will be ignored
- **field_names** (*Sequence[str], optional*) – An array of field names used as the *field_names* for the returned dict keys. If not given, column names will be used as the field_names.

Examples

```
>>> from hangar import Repository
>>> from torch.utils.data import DataLoader
>>> from hangar import make_torch_dataset
>>> repo = Repository('.')
>>> co = repo.checkout()
>>> aset = co.columns['dummy_aset']
>>> torch_dset = make_torch_dataset(aset, index_range=slice(1, 100))
>>> loader = DataLoader(torch_dset, batch_size=16)
>>> for batch in loader:
...     train_model(batch)
```

Returns

Return type torch.utils.data.Dataset

4.6 Hangar Tutorial

4.6.1 Quick Start Tutorial

A simple step-by-step guide that will quickly get you started with Hangar basics, including initializing a repository, adding and committing data to a repository.

Installation

You can install Hangar via pip:

```
$ pip install hangar
```

or via conda:

```
$ conda install -c conda-forge hangar
```

Please refer to the *Installation* page for more information.

Quick Start for the Impatient

The only import statement you'll ever need:

```
[1]: from hangar import Repository
```

Create and initialize a new Hangar Repository at the given path:

```
[2]: !mkdir /Volumes/Archivio/tensorwerk/hangar/quick-start

     repo = Repository(path="/Volumes/Archivio/tensorwerk/hangar/quick-start")

     repo.init(
         user_name="Alessia Marcolini", user_email="alessia@tensorwerk.com", remove_
      ↪old=True
     )
```

```
     Hangar Repo initialized at: /Volumes/Archivio/tensorwerk/hangar/quick-start/.hangar
```

```
     //anaconda/envs/hangar-tutorial/lib/python3.8/site-packages/hangar/context.py:92:␣
      ↪UserWarning: No repository exists at /Volumes/Archivio/tensorwerk/hangar/quick-
      ↪start/.hangar, please use `repo.init()` method
       warnings.warn(msg, UserWarning)
```

```
[2]: '/Volumes/Archivio/tensorwerk/hangar/quick-start/.hangar'
```

Checkout the Repository in write mode:

```
[3]: master_checkout = repo.checkout(write=True)
     master_checkout
```

```
[3]: Hangar WriterCheckout
         Writer       : True
         Base Branch  : master
         Num Columns  : 0
```

Inspect the columns we have (we just started, none so far):

```
[4]: master_checkout.columns
```

```
[4]: Hangar Columns
         Writeable         : True
         Number of Columns : 0
         Column Names / Partial Remote References:
           -
```

Prepare some random data to play with:

```
[5]: import numpy as np

     dummy = np.random.rand(3,2)
     dummy
```

```
[5]: array([[0.17961852, 0.31945355],
            [0.10929027, 0.2681622 ],
            [0.29397449, 0.02659856]])
```

Create a new column named dummy_column:

```
[6]: dummy_col = master_checkout.add_ndarray_column(name="dummy_column", prototype=dummy)
     dummy_col
```

```
[6]: Hangar FlatSampleWriter
         Column Name              : dummy_column
         Writeable                : True
         Column Type              : ndarray
         Column Layout            : flat
         Schema Type              : fixed_shape
         DType                    : float64
         Shape                    : (3, 2)
         Number of Samples        : 0
         Partial Remote Data Refs : False
```

Add data to `dummy_column`, treating it as a normal Python dictionary:

```
[7]: dummy_col[0] = dummy
```

```
[8]: dummy_col[1] = np.random.rand(3,2)
```

Commit your changes providing a message:

```
[9]: master_checkout.commit("Add dummy_column with 2 samples")
```

```
[9]: 'a=c104ef7e2cfe87318e78addd6033028488050cea'
```

Add more data and commit again:

```
[10]: dummy_col[2] = np.random.rand(3,2)
      dummy_col
```

```
[10]: Hangar FlatSampleWriter
          Column Name              : dummy_column
          Writeable                : True
          Column Type              : ndarray
          Column Layout            : flat
          Schema Type              : fixed_shape
          DType                    : float64
          Shape                    : (3, 2)
          Number of Samples        : 3
          Partial Remote Data Refs : False
```

```
[11]: master_checkout.commit("Add one more sample to dummy_column")
```

```
[11]: 'a=099557d48edebb7607fa3ec648eafa2a1af5e652'
```

See the master branch history:

```
[12]: master_checkout.log()
```

```
* a=099557d48edebb7607fa3ec648eafa2a1af5e652 (master) : Add one more sample to dummy_
↪column
* a=c104ef7e2cfe87318e78addd6033028488050cea : Add dummy_column with 2 samples
```

Close the write-enabled checkout:

[13]:
```
master_checkout.close()
```

Inspect the status of the `Repository`:

[14]:
```
repo.summary()
```

```
Summary of Contents Contained in Data Repository

==================
| Repository Info
|-----------------
|  Base Directory: /Volumes/Archivio/tensorwerk/hangar/quick-start
|  Disk Usage: 237.53 kB

===================
| Commit Details
-------------------
|  Commit: a=099557d48edebb7607fa3ec648eafa2a1af5e652
|  Created: Mon May  4 13:00:43 2020
|  By: Alessia Marcolini
|  Email: alessia@tensorwerk.com
|  Message: Add one more sample to dummy_column

==================
| DataSets
|-----------------
|  Number of Named Columns: 1
|
|  * Column Name: ColumnSchemaKey(column="dummy_column", layout="flat")
|    Num Data Pieces: 3
|    Details:
|    - column_layout: flat
|    - column_type: ndarray
|    - schema_hasher_tcode: 1
|    - data_hasher_tcode: 0
|    - schema_type: fixed_shape
|    - shape: (3, 2)
|    - dtype: float64
|    - backend: 01
|    - backend_options: {'complib': 'blosc:lz4hc', 'complevel': 5, 'shuffle': 'byte'}
```

Quick Start - with explanations

1. Create and initialize a "Repository"

Central to Hangar is the concept of *Repository*.

A `Repository` consists of an **historically ordered mapping** of **Commits** over time by various **Committers** across any number of **Branches**. Though there are many conceptual similarities in what a Git repo and a Hangar repository achieve, Hangar is designed with the express purpose of dealing with **numeric data**.

To start using Hangar programmatically, simply begin with this import statement:

[1]:
```
from hangar import Repository
```

Create the folder where you want to store the `Repository`:

```
[2]: !mkdir /Volumes/Archivio/tensorwerk/hangar/quick-start
```

Initialize the `Repository` object by saying where your repository should live.

Note: Note that if you feed a path to the `Repository` which does not contain a pre-initialized Hangar repo, Python shows you a warning saying that you will need to initialize the repo before starting working on it.

```
[3]: repo = Repository(path="/Volumes/Archivio/tensorwerk/hangar/quick-start")
```

```
//anaconda/envs/hangar-tutorial/lib/python3.8/site-packages/hangar/context.py:92:␣
↪UserWarning: No repository exists at /Volumes/Archivio/tensorwerk/hangar/quick-
↪start/.hangar, please use `repo.init()` method
  warnings.warn(msg, UserWarning)
```

Initialize the `Repository` providing your name and your email.

Warning: Please be aware that the `remove_old` parameter set to `True` **removes and reinitializes** a Hangar repository at the given path.

```
[4]: repo.init(
         user_name="Alessia Marcolini", user_email="alessia@tensorwerk.com", remove_
     ↪old=True
     )
```

```
Hangar Repo initialized at: /Volumes/Archivio/tensorwerk/hangar/quick-start/.hangar
```

```
[4]: '/Volumes/Archivio/tensorwerk/hangar/quick-start/.hangar'
```

2. Open the Staging Area for Writing

To start interacting with Hangar, first you need to check out the `Repository` you want to work on.

A repo can be checked out in two modes:

- *write-enabled*
- *read-only*

We need to check out the repo in **write mode** in order to initialize the columns and write into them.

```
[5]: master_checkout = repo.checkout(write=True)
     master_checkout
```

```
[5]: Hangar WriterCheckout
         Writer       : True
         Base Branch  : master
         Num Columns  : 0
```

A checkout allows access to `columns`. The `columns` attribute of a checkout provide the interface to working with all of the data on disk!

```
[6]: master_checkout.columns
```

[6]:
```
Hangar Columns
    Writeable         : True
    Number of Columns : 0
    Column Names / Partial Remote References:
      -
```

3. Create some random data to play with

Let's create a random array to be used as a dummy example:

[7]:
```
import numpy as np

dummy = np.random.rand(3,2)
dummy
```

[7]:
```
array([[0.54631485, 0.26578857],
       [0.74990074, 0.41764666],
       [0.75884524, 0.05547267]])
```

4. Initialize a column

With checkout write-enabled, we can now initialize a new column of the repository using the method *add_ndarray_column()*.

All samples within a column have the same data type, and number of dimensions. The size of each dimension can be either fixed (the default behavior) or variable per sample.

You will need to provide a column name and a prototype, so Hangar can infer the shape of the elements contained in the array. `dummy_col` will become a column accessor object.

[8]:
```
dummy_col = master_checkout.add_ndarray_column(name="dummy_column", prototype=dummy)
dummy_col
```

[8]:
```
Hangar FlatSampleWriter
    Column Name              : dummy_column
    Writeable                : True
    Column Type              : ndarray
    Column Layout            : flat
    Schema Type              : fixed_shape
    DType                    : float64
    Shape                    : (3, 2)
    Number of Samples        : 0
    Partial Remote Data Refs : False
```

Verify we successfully added the new column:

[9]:
```
master_checkout.columns
```

[9]:
```
Hangar Columns
    Writeable         : True
    Number of Columns : 1
    Column Names / Partial Remote References:
      - dummy_column / False
```

5. Add data

To add data to a named column, we can use **dict-style mode** as follows. Sample keys can be either str or int type.

```
[10]: dummy_col[0] = dummy
```

As we can see, `Number of Samples` is equal to 1 now!

```
[11]: dummy_col
```

```
[11]: Hangar FlatSampleWriter
          Column Name              : dummy_column
          Writeable                : True
          Column Type              : ndarray
          Column Layout            : flat
          Schema Type              : fixed_shape
          DType                    : float64
          Shape                    : (3, 2)
          Number of Samples        : 1
          Partial Remote Data Refs : False
```

```
[12]: dummy_col[1] = np.random.rand(3,2)
```

```
[13]: dummy_col
```

```
[13]: Hangar FlatSampleWriter
          Column Name              : dummy_column
          Writeable                : True
          Column Type              : ndarray
          Column Layout            : flat
          Schema Type              : fixed_shape
          DType                    : float64
          Shape                    : (3, 2)
          Number of Samples        : 2
          Partial Remote Data Refs : False
```

```
[14]: dummy_col[1]
```

```
[14]: array([[0.17590758, 0.26950355],
             [0.88036219, 0.7839301 ],
             [0.87321484, 0.04316646]])
```

You can also iterate over your column, as you would do with a regular Python dictionary:

```
[15]: for key, value in dummy_col.items():
          print('Key:', key)
          print('Value:', value)
          print()
```

```
Key: 0
Value: [[0.54631485 0.26578857]
 [0.74990074 0.41764666]
 [0.75884524 0.05547267]]

Key: 1
Value: [[0.17590758 0.26950355]
```

(continues on next page)

(continued from previous page)

```
 [0.88036219 0.7839301 ]
 [0.87321484 0.04316646]]
```

How many samples are in the column?

```
[16]: len(dummy_col)
```

```
[16]: 2
```

Does the column contain that key?

```
[17]: 0 in dummy_col
```

```
[17]: True
```

```
[18]: 5 in dummy_col
```

```
[18]: False
```

6. Commit changes

Once you have made a set of changes you want to **commit**, just simply call the *commit()* method (and pass in a message)!

```
[19]: master_checkout.commit("Add dummy_column with 2 samples")
```

```
[19]: 'a=4f42fce2b66476271f149e3cd2eb4c6ba66daeee'
```

Let's add another sample in the column:

```
[20]: dummy_col[2] = np.random.rand(3,2)
      dummy_col
```

```
[20]: Hangar FlatSampleWriter
          Column Name             : dummy_column
          Writeable               : True
          Column Type             : ndarray
          Column Layout           : flat
          Schema Type             : fixed_shape
          DType                   : float64
          Shape                   : (3, 2)
          Number of Samples       : 3
          Partial Remote Data Refs : False
```

`Number of Samples` is equal to 3 now and we want to keep track of the change with another commit:

```
[21]: master_checkout.commit("Add one more sample to dummy_column")
```

```
[21]: 'a=753e28e27d4b23a0dca0633f90b4513538a98c40'
```

To view the **history** of your commits:

```
[22]: master_checkout.log()
```

```
* a=753e28e27d4b23a0dca0633f90b4513538a98c40 (master) : Add one more sample to dummy_
↪column
* a=4f42fce2b66476271f149e3cd2eb4c6ba66daeee : Add dummy_column with 2 samples
```

Do not forget to close the write-enabled checkout!

[23]:
```
master_checkout.close()
```

Check the **state of the repository** and get useful information about disk usage, the columns you have and the last commit:

[24]:
```
repo.summary()
```

```
Summary of Contents Contained in Data Repository

==================
| Repository Info
|-----------------
|  Base Directory: /Volumes/Archivio/tensorwerk/hangar/quick-start
|  Disk Usage: 237.53 kB

===================
| Commit Details
-------------------
|  Commit: a=753e28e27d4b23a0dca0633f90b4513538a98c40
|  Created: Tue Apr 21 21:50:15 2020
|  By: Alessia Marcolini
|  Email: alessia@tensorwerk.com
|  Message: Add one more sample to dummy_column

==================
| DataSets
|-----------------
|  Number of Named Columns: 1
|
|  * Column Name: ColumnSchemaKey(column="dummy_column", layout="flat")
|    Num Data Pieces: 3
|    Details:
|    - column_layout: flat
|    - column_type: ndarray
|    - schema_hasher_tcode: 1
|    - data_hasher_tcode: 0
|    - schema_type: fixed_shape
|    - shape: (3, 2)
|    - dtype: float64
|    - backend: 01
|    - backend_options: {'complib': 'blosc:lz4hc', 'complevel': 5, 'shuffle': 'byte'}
```

4.6.2 Part 1: Creating A Repository And Working With Data

This tutorial will review the first steps of working with a hangar repository.

To fit with the beginner's theme, we will use the MNIST dataset. Later examples will show off how to work with much more complex data.

[1]:

```
from hangar import Repository

import numpy as np
import pickle
import gzip
import matplotlib.pyplot as plt

from tqdm import tqdm
```

Creating & Interacting with a Hangar Repository

Hangar is designed to "just make sense" in every operation you have to perform. As such, there is a single interface which all interaction begins with: the designed to "just make sense" in every operation you have to perform. As such, there is a single interface which all interaction begins with: the *Repository* object.

Whether a hangar repository exists at the path you specify or not, just tell hangar where it should live!

Intitializing a repository

The first time you want to work with a new repository, the repository *init()* method must be called. This is where you provide Hangar with your name and email address (to be used in the commit log), as well as implicitly confirming that you do want to create the underlying data files hangar uses on disk.

[2]:

```
repo = Repository(path='/Users/rick/projects/tensorwerk/hangar/dev/mnist/')

# First time a repository is accessed only!
# Note: if you feed a path to the `Repository` which does not contain a pre-
↪initialized hangar repo,
# when the Repository object is initialized it will let you know that you need to run␣
↪`init()`

repo.init(user_name='Rick Izzo', user_email='rick@tensorwerk.com', remove_old=True)
```

```
Hangar Repo initialized at: /Users/rick/projects/tensorwerk/hangar/dev/mnist/.hangar
```

[2]:

```
'/Users/rick/projects/tensorwerk/hangar/dev/mnist/.hangar'
```

Checking out the repo for writing

A repository can be checked out in two modes:

1. *write-enabled*: applies all operations to the staging area's current state. Only one write-enabled checkout can be active at a different time, must be closed upon last use, or manual intervention will be needed to remove the writer lock.
2. *read-only*: checkout a commit or branch to view repository state as it existed at that point in time.

Lots of useful information is in the iPython `__repr__`

If you're ever in doubt about what the state of the object your working on is, just call its reps, and the most relevant information will be sent to your screen!

```
[3]: co = repo.checkout(write=True)
     co
```

```
[3]: Hangar WriterCheckout
         Writer       : True
         Base Branch  : master
         Num Columns  : 0
```

A checkout allows access to columns

The *columns* attributes of a checkout provide the interface to working with all of the data on disk!

```
[4]: co.columns
```

```
[4]: Hangar Columns
         Writeable         : True
         Number of Columns : 0
         Column Names / Partial Remote References:
           -
```

Before data can be added to a repository, a column must be initialized.

We're going to first load up a the MNIST pickled dataset so it can be added to the repo!

```
[6]: # Load the dataset
     with gzip.open('/Users/rick/projects/tensorwerk/hangar/dev/data/mnist.pkl.gz', 'rb')␣
     ↪as f:
         train_set, valid_set, test_set = pickle.load(f, encoding='bytes')

     def rescale(array):
         array = array * 256
         rounded = np.round(array)
         return rounded.astype(np.uint8())

     sample_trimg = rescale(train_set[0][0])
     sample_trlabel = np.array([train_set[1][0]])
     trimgs = rescale(train_set[0])
     trlabels = train_set[1]
```

Before data can be added to a repository, a column must be initialized.

An "Column" is a named grouping of data samples where each sample shares a number of similar attributes and array properties.

See the docstrings below or in *add_ndarray_column()*

`WriterCheckout.`**`add_ndarray_column`**(*name: str, shape: Optional[Union[int, tuple]] = None, dtype: Optional[numpy.dtype] = None, prototype: Optional[numpy.ndarray] = None, variable_shape: bool = False, contains_subsamples: bool = False, *, backend: Optional[str] = None, backend_options: Optional[dict] = None*)

Initializes a `numpy.ndarray` container column.

Columns are created in order to store some arbitrary collection of data pieces. In this case, we store `numpy.ndarray` data. Items need not be related to each-other in any direct capacity; the only criteria hangar requires is that all pieces of data stored in the column have a compatible schema with each-other (more on this below). Each piece of data is indexed by some key (either user defined or automatically generated depending on the user's preferences). Both single level stores (sample keys mapping to data on disk) and nested stores (where some sample key maps to an arbitrary number of subsamples, in turn each pointing to some piece of store data on disk) are supported.

All data pieces within a column have the same data type and number of dimensions. The size of each dimension can be either fixed (the default behavior) or variable per sample. For fixed dimension sizes, all data pieces written to the column must have the same shape & size which was specified at the time the column column was initialized. Alternatively, variable sized columns can write data pieces with dimensions of any size (up to a specified maximum).

Parameters

- **`name`** (*`str`*) – The name assigned to this column.
- **`shape`** (*`Optional[Union[int, Tuple[int]]]`*) – The shape of the data samples which will be written in this column. This argument and the *dtype* argument are required if a *prototype* is not provided, defaults to None.
- **`dtype`** (Optional[`numpy.dtype`]) – The datatype of this column. This argument and the *shape* argument are required if a *prototype* is not provided., defaults to None.
- **`prototype`** (Optional[`numpy.ndarray`]) – A sample array of correct datatype and shape which will be used to initialize the column storage mechanisms. If this is provided, the *shape* and *dtype* arguments must not be set, defaults to None.
- **`variable_shape`** (*`bool, optional`*) – If this is a variable sized column. If true, a the maximum shape is set from the provided `shape` or `prototype` argument. Any sample added to the column can then have dimension sizes <= to this initial specification (so long as they have the same rank as what was specified) defaults to False.
- **`contains_subsamples`** (*`bool, optional`*) – True if the column column should store data in a nested structure. In this scheme, a sample key is used to index an arbitrary number of subsamples which map some (sub)key to some piece of data. If False, sample keys map directly to a single piece of data; essentially acting as a single level key/value store. By default, False.
- **`backend`** (*`Optional[str], optional`*) – ADVANCED USERS ONLY, backend format code to use for column data. If None, automatically inferred and set based on data shape and type. by default None

- **backend_options** (*Optional[dict], optional*) – ADVANCED USERS ONLY, filter opts to apply to column data. If None, automatically inferred and set based on data shape and type. by default None

Returns instance object of the initialized column.

Return type `Columns`

WriterCheckout.**add_str_column**(*name: str, contains_subsamples: bool = False, *, backend: Optional[str] = None, backend_options: Optional[dict] = None*)

Initializes a `str` container column

Columns are created in order to store some arbitrary collection of data pieces. In this case, we store `str` data. Items need not be related to each-other in any direct capacity; the only criteria hangar requires is that all pieces of data stored in the column have a compatible schema with each-other (more on this below). Each piece of data is indexed by some key (either user defined or automatically generated depending on the user's preferences). Both single level stores (sample keys mapping to data on disk) and nested stores (where some sample key maps to an arbitrary number of subsamples, in turn each pointing to some piece of store data on disk) are supported.

All data pieces within a column have the same data type. For `str` columns, there is no distinction between `'variable_shape'` and `'fixed_shape'` schema types. Values are allowed to take on a value of any size so long as the datatype and contents are valid for the schema definition.

Parameters

- **name** (*str*) – Name assigned to the column
- **contains_subsamples** (*bool, optional*) – True if the column column should store data in a nested structure. In this scheme, a sample key is used to index an arbitrary number of subsamples which map some (sub)key to a piece of data. If False, sample keys map directly to a single piece of data; essentially acting as a single level key/value store. By default, False.
- **backend** (*Optional[str], optional*) – ADVANCED USERS ONLY, backend format code to use for column data. If None, automatically inferred and set based on data shape and type. by default None
- **backend_options** (*Optional[dict], optional*) – ADVANCED USERS ONLY, filter opts to apply to column data. If None, automatically inferred and set based on data shape and type. by default None

Returns instance object of the initialized column.

Return type `Columns`

WriterCheckout.**add_bytes_column**(*name: str, contains_subsamples: bool = False, *, backend: Optional[str] = None, backend_options: Optional[dict] = None*)

Initializes a `bytes` container column

Columns are created in order to store some arbitrary collection of data pieces. In this case, we store `bbytes` data. Items need not be related to each-other in any direct capacity; the only criteria hangar requires is that all pieces of data stored in the column have a compatible schema with each-other (more on this below). Each piece of data is indexed by some key (either user defined or automatically generated depending on the user's preferences). Both single level stores (sample keys mapping to data on disk) and nested stores (where some sample key maps to an arbitrary number of subsamples, in turn each pointing to some piece of store data on disk) are supported.

All data pieces within a column have the same data type. For `bytes` columns, there is no distinction between `'variable_shape'` and `'fixed_shape'` schema types. Values are allowed to take on a value of any size so long as the datatype and contents are valid for the schema definition.

Parameters

- **name** (*str*) – Name assigned to the column
- **contains_subsamples** (*bool, optional*) – True if the column column should store data in a nested structure. In this scheme, a sample key is used to index an arbitrary number of subsamples which map some (sub)key to a piece of data. If False, sample keys map directly to a single piece of data; essentially acting as a single level key/value store. By default, False.
- **backend** (*Optional[str], optional*) – ADVANCED USERS ONLY, backend format code to use for column data. If None, automatically inferred and set based on data shape and type. by default None
- **backend_options** (*Optional[dict], optional*) – ADVANCED USERS ONLY, filter opts to apply to column data. If None, automatically inferred and set based on data shape and type. by default None

Returns instance object of the initialized column.

Return type *Columns*

```
[7]: col = co.add_ndarray_column(name='mnist_training_images', prototype=trimgs[0])
```

```
[8]: col
```

```
[8]: Hangar FlatSampleWriter
         Column Name              : mnist_training_images
         Writeable                : True
         Column Type              : ndarray
         Column Layout            : flat
         Schema Type              : fixed_shape
         DType                    : uint8
         Shape                    : (784,)
         Number of Samples        : 0
         Partial Remote Data Refs : False
```

Interaction

Through columns attribute

When a column is initialized, a column accessor object will be returned, however, depending on your use case, this may or may not be the most convenient way to access a arrayset.

In general, we have implemented a full `dict` mapping interface on top of all objects. To access the `'mnist_training_images'` arrayset you can just use a dict style access like the following (note: if operating in iPython/Jupyter, the arrayset keys will autocomplete for you).

The column objects returned here contain many useful instrospecion methods which we will review over the rest of the tutorial.

```
[9]: co.columns['mnist_training_images']
```

```
[9]: Hangar FlatSampleWriter
         Column Name              : mnist_training_images
         Writeable                : True
         Column Type              : ndarray
         Column Layout            : flat
         Schema Type              : fixed_shape
```

(continues on next page)

(continued from previous page)

```
    DType                         : uint8
    Shape                         : (784,)
    Number of Samples             : 0
    Partial Remote Data Refs : False
```

[10]:

```
train_aset = co.columns['mnist_training_images']

# OR an equivalent way using the `.get()` method

train_aset = co.columns.get('mnist_training_images')
train_aset
```

[10]:

```
Hangar FlatSampleWriter
    Column Name                   : mnist_training_images
    Writeable                     : True
    Column Type                   : ndarray
    Column Layout                 : flat
    Schema Type                   : fixed_shape
    DType                         : uint8
    Shape                         : (784,)
    Number of Samples             : 0
    Partial Remote Data Refs : False
```

Through the checkout object (arrayset and sample access)

In addition to the standard `co.columns` access methods, we have implemented a convenience mapping to *columns* and *flat samples* or *nested samples* / *nested subsamples* (ie. data) for both reading and writing from the *checkout* object itself.

To get the same arrayset object from the checkout, simply use:

[11]:

```
train_asets = co['mnist_training_images']
train_asets
```

[11]:

```
Hangar FlatSampleWriter
    Column Name                   : mnist_training_images
    Writeable                     : True
    Column Type                   : ndarray
    Column Layout                 : flat
    Schema Type                   : fixed_shape
    DType                         : uint8
    Shape                         : (784,)
    Number of Samples             : 0
    Partial Remote Data Refs : False
```

Though that works as expected, most use cases will take advantage of adding and reading data from multiple columns / samples at a time. This is shown in the next section.

Adding Data

To add data to a named arrayset, we can use dict-style setting (refer to the `__setitem__`, `__getitem__`, and `__delitem__` methods), or the `update()` method. Sample keys can be either `str` or `int` type.

[12]:
```
train_aset['0'] = trimgs[0]

data = {
    '1': trimgs[1],
    '2': trimgs[2],
}
train_aset.update(data)

train_aset[51] = trimgs[51]
```

Using the checkout method

[13]:
```
co['mnist_training_images', 60] = trimgs[60]
```

How many samples are in the arrayset?

[14]:
```
len(train_aset)
```

[14]:
```
5
```

Containment Testing

[15]:
```
'hi' in train_aset
```

[15]:
```
False
```

[16]:
```
'0' in train_aset
```

[16]:
```
True
```

[17]:
```
60 in train_aset
```

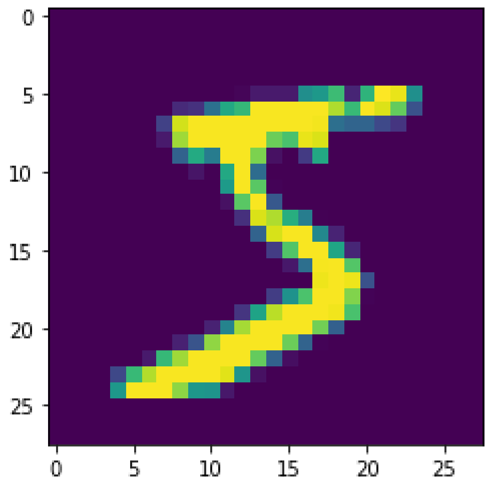
[17]:
```
True
```

Dictionary Style Retrieval for known keys

[18]:
```
out1 = train_aset['0']
# OR
out2 = co['mnist_training_images', '0']

print(np.allclose(out1, out2))

plt.imshow(out1.reshape(28, 28))
```

```
True
```

[18]:
```
<matplotlib.image.AxesImage at 0x3703cc7f0>
```

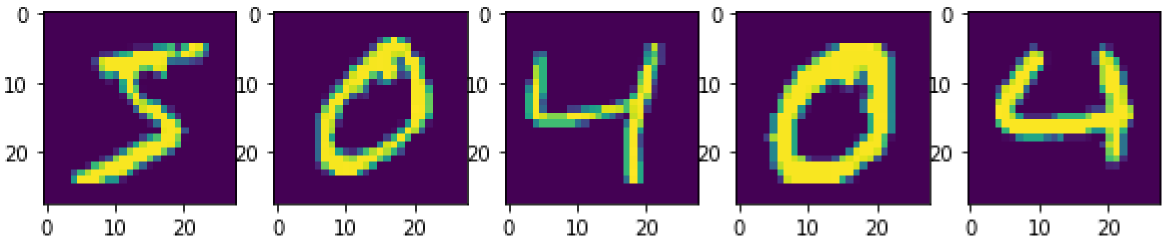


Dict style iteration supported out of the box

[19]:

```
# iterate normally over keys

for k in train_aset:
    # equivalent method: for k in train_aset.keys():
    print(k)

# iterate over items (plot results)

fig, axs = plt.subplots(nrows=1, ncols=5, figsize=(10, 10))

for idx, v in enumerate(train_aset.values()):
    axs[idx].imshow(v.reshape(28, 28))
plt.show()

# iterate over items, store k, v in dict

myDict = {}
for k, v in train_aset.items():
    myDict[k] = v
```

```
0
1
2
51
60
```



Performance

Once you've completed an interactive exploration, be sure to use the context manager form of the `update()` and `get()` methods!

In order to make sure that all your data is always safe in Hangar, the backend diligently ensures that all contexts (operations which can somehow interact with the record structures) are opened and closed appropriately. When you use the context manager form of a arrayset object, we can offload a significant amount of work to the python runtime, and dramatically increase read and write speeds.

Most columns we've tested see an increased throughput differential of 250% - 500% for writes and 300% - 600% for reads when comparing using the context manager form vs the naked form!

[23]:

```
import time

# ----------------- Non Context Manager Form ----------------------

co = repo.checkout(write=True)
aset_trimgs = co.add_ndarray_column(name='train_images', prototype=sample_trimg)
aset_trlabels = co.add_ndarray_column(name='train_labels', prototype=sample_trlabel)

print(f'Beginning non-context manager form')
print('----------------------------------')
start_time = time.time()

for idx, img in enumerate(trimgs):
    aset_trimgs[idx] = img
    aset_trlabels[idx] = np.array([trlabels[idx]])

print(f'Finished non-context manager form in: {time.time() - start_time} seconds')

co.reset_staging_area()
co.close()

# ----------------- Context Manager Form --------------------------

co = repo.checkout(write=True)
aset_trimgs = co.add_ndarray_column(name='train_images', prototype=sample_trimg)
aset_trlabels = co.add_ndarray_column(name='train_labels', prototype=sample_trlabel)

print(f'\nBeginning context manager form')
print('--------------------------------')
start_time = time.time()

with aset_trimgs, aset_trlabels:
    for idx, img in enumerate(trimgs):
        aset_trimgs[idx] = img
        aset_trlabels[idx] = np.array([trlabels[idx]])

print(f'Finished context manager form in: {time.time() - start_time} seconds')

co.reset_staging_area()
co.close()

print(f'Finished context manager with checkout form in: {time.time() - start_time}␣
↪seconds')
```

```
Beginning non-context manager form
----------------------------------
```

(continues on next page)

(continued from previous page)

```
Finished non-context manager form in: 78.54769086837769 seconds
Hard reset requested with writer_lock: 8910b50e-1f9d-4cb1-986c-b99ea84c8a54

Beginning context manager form
--------------------------------
Finished context manager form in: 11.608536720275879 seconds
Hard reset requested with writer_lock: ad4a2ef9-8494-49f8-84ef-40c3990b1e9b
```

Clearly, the context manager form is far and away superior, however we fell that for the purposes of interactive use that the "Naked" form is valubal to the average user!

Commiting Changes

Once you have made a set of changes you want to commit, just simply call the *commit()* method (and pass in a message)!

```
[25]: co.commit('hello world, this is my first hangar commit')
```

```
[25]: 'a=8eb01eaf0c657f8526dbf9a8ffab0a4606ebfd3b'
```

The returned value (`'e11d061dc457b361842801e24cbd119a745089d6'`) is the commit hash of this commit. It may be useful to assign this to a variable and follow this up by creating a branch from this commit!

Don't Forget to Close the Write-Enabled Checkout to Release the Lock!

We mentioned in `Checking out the repo for writing` that when a `write-enabled` checkout is created, it places a lock on writers until it is closed. If for whatever reason the program terminates via a non python `SIGKILL` or fatal interpreter error without closing the write-enabled checkout, this lock will persist (forever technically, but realistically until it is manually freed).

Luckily, preventing this issue from occurring is as simple as calling *close()*!

If you forget, normal interperter shutdown should trigger an `atexit` hook automatically, however this behavior should not be relied upon. Is better to just call *close()*.

```
[26]: co.close()
```

But if you did forget, and you recieve a `PermissionError` next time you open a checkout

```
PermissionError: Cannot acquire the writer lock. Only one instance of
a writer checkout can be active at a time. If the last checkout of this
repository did not properly close, or a crash occured, the lock must be
manually freed before another writer can be instantiated.
```

You can manually free the lock with the following method. However!

This is a dangerous operation, and it's one of the only ways where a user can put data in their repository at risk! If another python process is still holding the lock, do NOT force the release. Kill the process (that's totally fine to do at any time, then force the lock release).

```
[27]: repo.force_release_writer_lock()
```

[27]:

```
True
```

Reading Data

Two different styles of access are considered below, In general, the contex manager form if recomended (though marginal performance improvements are expected to be seen at best)

[29]:

```
co = repo.checkout()

trlabel_col = co['train_labels']
trimg_col = co['train_images']

print(f'\nBegining Key Iteration')
print('-----------------------')
start = time.time()

for idx in trimg_col.keys():
    image_data = trimg_col[idx]
    label_data = trlabel_col[idx]

print(f'completed in {time.time() - start} sec')

print(f'\nBegining Items Iteration with Context Manager')
print('---------------------------------------------')
start = time.time()

with trlabel_col, trimg_col:
    for index, image_data in trimg_col.items():
        label_data = trlabel_col[index]

print(f'completed in {time.time() - start} sec')

co.close()
```

```
 Neither BRANCH or COMMIT specified.
 * Checking out writing HEAD BRANCH: master

Begining Key Iteration
-----------------------
completed in 5.838773965835571 sec

Begining Items Iteration with Context Manager
---------------------------------------------
completed in 5.516948938369751 sec
```

Inspecting state from the top!

After your first commit, the summary and log methods will begin to work, and you can either print the stream to the console (as shown below), or you can dig deep into the internal of how hangar thinks about your data! (To be covered in an advanced tutorial later on).

The point is, regardless of your level of interaction with a live hangar repository, all level of state is accessable from the top, and in general has been built to be the only way to directly access it!

```
[30]: repo.summary()
```

```
Summary of Contents Contained in Data Repository

==================
| Repository Info
|-----------------
|  Base Directory: /Users/rick/projects/tensorwerk/hangar/dev/mnist
|  Disk Usage: 57.29 MB

===================
| Commit Details
-------------------
|  Commit: a=8eb01eaf0c657f8526dbf9a8ffab0a4606ebfd3b
|  Created: Tue Feb 25 19:03:06 2020
|  By: Rick Izzo
|  Email: rick@tensorwerk.com
|  Message: hello world, this is my first hangar commit

==================
| DataSets
|-----------------
|  Number of Named Columns: 2
|
|  * Column Name: ColumnSchemaKey(column="train_images", layout="flat")
|    Num Data Pieces: 50000
|    Details:
|    - column_layout: flat
|    - column_type: ndarray
|    - schema_type: fixed_shape
|    - shape: (784,)
|    - dtype: uint8
|    - backend: 00
|    - backend_options: {'complib': 'blosc:lz4hc', 'complevel': 5, 'shuffle': 'byte'}
|
|  * Column Name: ColumnSchemaKey(column="train_labels", layout="flat")
|    Num Data Pieces: 50000
|    Details:
|    - column_layout: flat
|    - column_type: ndarray
|    - schema_type: fixed_shape
|    - shape: (1,)
|    - dtype: int64
|    - backend: 10
|    - backend_options: {}

==================
| Metadata:
|-----------------
```

(continues on next page)

(continued from previous page)

```
|  Number of Keys: 0
```

[31]:

```
repo.log()
```

```
* a=8eb01eaf0c657f8526dbf9a8ffab0a4606ebfd3b (master) : hello world, this is my first␣
↪hangar commit
```

4.6.3 Part 2: Checkouts, Branching, & Merging

This section deals with navigating repository history, creating & merging branches, and understanding conflicts.

The Hangar Workflow

The hangar workflow is intended to mimic common `git` workflows in which small incremental changes are made and committed on dedicated `topic` branches. After the `topic` has been adequatly set, `topic` branch is merged into a separate branch (commonly referred to as `master`, though it need not to be the actual branch named `"master"`), where well vetted and more permanent changes are kept.

```
Create Branch -> Checkout Branch -> Make Changes -> Commit
```

Making the Initial Commit

Let's initialize a new repository and see how branching works in Hangar:

[1]:

```
from hangar import Repository
import numpy as np
```

[2]:

```
repo = Repository(path='/Users/rick/projects/tensorwerk/hangar/dev/mnist/')
```

[3]:

```
repo_pth = repo.init(user_name='Test User', user_email='test@foo.com', remove_
↪old=True)
```

```
Hangar Repo initialized at: /Users/rick/projects/tensorwerk/hangar/dev/mnist/.hangar
```

When a repository is first initialized, it has no history, no commits.

[4]:

```
repo.log() # -> returns None
```

Though the repository is essentially empty at this point in time, there is one thing which is present: a branch with the name: `"master"`.

[5]:

```
repo.list_branches()
```

[5]:

```
['master']
```

This `"master"` is the branch we make our first commit on; until we do, the repository is in a semi-unstable state; with no history or contents, most of the functionality of a repository (to store, retrieve, and work with versions of data across time) just isn't possible. A significant portion of otherwise standard operations will generally flat out refuse to execute (ie. read-only checkouts, log, push, etc.) until the first commit is made.

One of the only options available at this point is to create a write-enabled checkout on the `"master"` branch and to begin to add data so we can make a commit. Let's do that now:

```
[6]: co = repo.checkout(write=True)
```

As expected, there are no columns nor metadata samples recorded in the checkout.

```
[7]: print(f'number of metadata keys: {len(co.metadata)}')
     print(f'number of columns: {len(co.columns)}')
```

```
number of metadata keys: 0
number of columns: 0
```

Let's add a dummy array just to put something in the repository history to commit. We'll then close the checkout so we can explore some useful tools which depend on having at least one historical record (commit) in the repo.

```
[8]: dummy = np.arange(10, dtype=np.uint16)
     col = co.add_ndarray_column('dummy_column', prototype=dummy)
     col['0'] = dummy
     initialCommitHash = co.commit('first commit with a single sample added to a dummy␣
     ↪column')
     co.close()
```

If we check the history now, we can see our first commit hash, and that it is labeled with the branch name "master"

```
[9]: repo.log()
```

```
* a=eaee002ed9c6e949c3657bd50e3949d6a459d50e (master) : first commit with a single␣
↪sample added to a dummy column
```

So now our repository contains: - *A commit*: a fully independent description of the entire repository state as it existed at some point in time. A commit is identified by a `commit_hash`. - *A branch*: a label pointing to a particular `commit` / `commit_hash`.

Once committed, it is not possible to remove, modify, or otherwise tamper with the contents of a commit in any way. It is a permanent record, which Hangar has no method to change once written to disk.

In addition, as a `commit_hash` is not only calculated from the `commit` 's contents, but from the `commit_hash` of its parents (more on this to follow), knowing a single top-level `commit_hash` allows us to verify the integrity of the entire repository history. This fundamental behavior holds even in cases of disk-corruption or malicious use.

Working with Checkouts & Branches

As mentioned in the first tutorial, we work with the data in a repository through a *checkout*. There are two types of checkouts (each of which have different uses and abilities):

`Checking out a branch / commit for reading: <api.rst#read-only-checkout>`__ is the process of retrieving records describing repository state at some point in time, and setting up access to the referenced data.

- Any number of read checkout processes can operate on a repository (on any number of commits) at the same time.

`Checking out a branch for writing: <api.rst#write-enabled-checkout>`__ is the process of setting up a (mutable) `staging area` to temporarily gather record references / data before all changes have been made and staging area contents are committed in a new permanent record of history (a `commit`).

- Only one write-enabled checkout can ever be operating in a repository at a time.
- When initially creating the checkout, the `staging area` is not actually "empty". Instead, it has the full contents of the last `commit` referenced by a branch's `HEAD`. These records can be removed / mutated / added to in any way to form the next `commit`. The new `commit` retains a permanent reference identifying the previous `HEAD` `commit` was used as its base `staging area`.

- On commit, the branch which was checked out has its HEAD pointer value updated to the new commit's commit_hash. A write-enabled checkout starting from the same branch will now use that commit's record content as the base for its staging area.

Creating a branch

A branch is an individual series of changes / commits which diverge from the main history of the repository at some point in time. All changes made along a branch are completely isolated from those on other branches. After some point in time, changes made in a disparate branches can be unified through an automatic merge process (described in detail later in this tutorial). In general, the Hangar branching model is semantically identical to the Git one; The one exception is that in Hangar, a branch must always have a name and a base_commit. (No "Detached HEAD state" is possible for a write-enabled checkout). If No base_commit is specified, the current writer branch HEAD commit is used as the base_commit hash for the branch automatically.

Hangar branches have the same lightweight and performant properties which make working with Git branches so appealing - they are cheap and easy to use, create, and discard (if necessary).

To create a branch, use the *create_branch()* method.

```
[10]: branch_1 = repo.create_branch(name='testbranch')
```

```
[11]: branch_1
```

```
[11]: BranchHead(name='testbranch', digest='a=eaee002ed9c6e949c3657bd50e3949d6a459d50e')
```

We use the *list_branches()* and *log()* methods to see that a new branch named testbranch has been created and is indeed pointing to our initial commit.

```
[12]: print(f'branch names: {repo.list_branches()} \n')
      repo.log()
```

```
branch names: ['master', 'testbranch']

* a=eaee002ed9c6e949c3657bd50e3949d6a459d50e (master) (testbranch) : first commit␣
↪with a single sample added to a dummy column
```

If instead, we actually specify the base commit (with a different branch name) we see we do actually get a third branch. pointing to the same commit as master and testbranch

```
[13]: branch_2 = repo.create_branch(name='new', base_commit=initialCommitHash)
```

```
[14]: branch_2
```

```
[14]: BranchHead(name='new', digest='a=eaee002ed9c6e949c3657bd50e3949d6a459d50e')
```

```
[15]: repo.log()
```

```
* a=eaee002ed9c6e949c3657bd50e3949d6a459d50e (master) (new) (testbranch) : first␣
↪commit with a single sample added to a dummy column
```

Making changes on a branch

Let’s make some changes on the `new` branch to see how things work.

We can see that the data we added previously is still here (`dummy` arrayset containing one sample labeled `0`).

```
[16]: co = repo.checkout(write=True, branch='new')
```

```
[17]: co.columns
```

```
[17]: Hangar Columns
          Writeable         : True
          Number of Columns : 1
          Column Names / Partial Remote References:
            - dummy_column / False
```

```
[18]: co.columns['dummy_column']
```

```
[18]: Hangar FlatSampleWriter
          Column Name              : dummy_column
          Writeable                : True
          Column Type              : ndarray
          Column Layout            : flat
          Schema Type              : fixed_shape
          DType                    : uint16
          Shape                    : (10,)
          Number of Samples        : 1
          Partial Remote Data Refs : False
```

```
[19]: co.columns['dummy_column']['0']
```

```
[19]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint16)
```

Let’s add another sample to the `dummy_arrayset` called `1`

```
[20]: arr = np.arange(10, dtype=np.uint16)
      # let's increment values so that `0` and `1` aren't set to the same thing
      arr += 1

      co['dummy_column', '1'] = arr
```

We can see that in this checkout, there are indeed two samples in the `dummy_arrayset`:

```
[21]: len(co.columns['dummy_column'])
```

```
[21]: 2
```

That’s all, let’s commit this and be done with this branch.

```
[22]: co.commit('commit on `new` branch adding a sample to dummy_arrayset')
      co.close()
```

How do changes appear when made on a branch?

If we look at the log, we see that the branch we were on (`new`) is a commit ahead of `master` and `testbranch`

```
[23]: repo.log()
```

```
* a=c1cf1bd6863ed0b95239d2c9e1a6c6cc65569e94 (new) : commit on `new` branch adding a␣
↪sample to dummy_arrayset
* a=eaee002ed9c6e949c3657bd50e3949d6a459d50e (master) (testbranch) : first commit␣
↪with a single sample added to a dummy column
```

The meaning is exactly what one would intuit. We made some changes, they were reflected on the `new` branch, but the `master` and `testbranch` branches were not impacted at all, nor were any of the commits!

Merging (Part 1) Fast-Forward Merges

Say we like the changes we made on the `new` branch so much that we want them to be included into our `master` branch! How do we make this happen for this scenario??

Well, the history between the `HEAD` of the `new` and the `HEAD` of the `master` branch is perfectly linear. In fact, when we began making changes on `new`, our staging area was *identical* to what the `master HEAD` commit references are right now!

If you'll remember that a branch is just a pointer which assigns some `name` to a `commit_hash`, it becomes apparent that a merge in this case really doesn't involve any work at all. With a linear history between `master` and `new`, any `commits` exsting along the path between the `HEAD` of `new` and `master` are the only changes which are introduced, and we can be sure that this is the only view of the data records which can exist!

What this means in practice is that for this type of merge, we can just update the `HEAD` of `master` to point to the `HEAD` of `"new"`, and the merge is complete.

This situation is referred to as a **Fast Forward (FF) Merge**. A FF merge is safe to perform any time a linear history lies between the `HEAD` of some `topic` and `base` branch, regardless of how many commits or changes which were introduced.

For other situations, a more complicated **Three Way Merge** is required. This merge method will be explained a bit more later in this tutorial.

```
[24]: co = repo.checkout(write=True, branch='master')
```

Performing the Merge

In practice, you'll never need to know the details of the merge theory explained above (or even remember it exists). Hangar automatically figures out which merge algorithms should be used and then performed whatever calculations are needed to compute the results.

As a user, merging in Hangar is a one-liner! just use the *merge()* method from a `write-enabled` checkout (shown below), or the analogous methods method from the Repository Object *repo.merge()* (if not already working with a `write-enabled` checkout object).

```
[25]: co.merge(message='message for commit (not used for FF merge)', dev_branch='new')
```

```
Selected Fast-Forward Merge Strategy
```

```
[25]: 'a=c1cf1bd6863ed0b95239d2c9e1a6c6cc65569e94'
```

Let's check the log!

[26]:
```
repo.log()
```

```
* a=c1cf1bd6863ed0b95239d2c9e1a6c6cc65569e94 (master) (new) : commit on `new` branch␣
↪adding a sample to dummy_arrayset
* a=eaee002ed9c6e949c3657bd50e3949d6a459d50e (testbranch) : first commit with a␣
↪single sample added to a dummy column
```

[27]:
```
co.branch_name
```

[27]:
```
'master'
```

[28]:
```
co.commit_hash
```

[28]:
```
'a=c1cf1bd6863ed0b95239d2c9e1a6c6cc65569e94'
```

[29]:
```
co.columns['dummy_column']
```

[29]:
```
Hangar FlatSampleWriter
    Column Name              : dummy_column
    Writeable                : True
    Column Type              : ndarray
    Column Layout            : flat
    Schema Type              : fixed_shape
    DType                    : uint16
    Shape                    : (10,)
    Number of Samples        : 2
    Partial Remote Data Refs : False
```

As you can see, everything is as it should be!

[30]:
```
co.close()
```

Making changes to introduce diverged histories

Let's now go back to our `testbranch` branch and make some changes there so we can see what happens when changes don't follow a linear history.

[31]:
```
co = repo.checkout(write=True, branch='testbranch')
```

[32]:
```
co.columns
```

[32]:
```
Hangar Columns
    Writeable         : True
    Number of Columns : 1
    Column Names / Partial Remote References:
      - dummy_column / False
```

[33]:
```
co.columns['dummy_column']
```

[33]:
```
Hangar FlatSampleWriter
    Column Name              : dummy_column
    Writeable                : True
    Column Type              : ndarray
    Column Layout            : flat
    Schema Type              : fixed_shape
```

(continues on next page)

(continued from previous page)

```
    DType                       : uint16
    Shape                       : (10,)
    Number of Samples           : 1
    Partial Remote Data Refs : False
```

We will start by mutating sample `0` in `dummy_arrayset` to a different value

```
[34]: old_arr = co['dummy_column', '0']
      new_arr = old_arr + 50
      new_arr
```

```
[34]: array([50, 51, 52, 53, 54, 55, 56, 57, 58, 59], dtype=uint16)
```

```
[35]: co['dummy_column', '0'] = new_arr
```

Let's make a commit here, then add some metadata and make a new commit (all on the `testbranch` branch).

```
[36]: co.commit('mutated sample `0` of `dummy_column` to new value')
```

```
[36]: 'a=fcd82f86e39b19c3e5351dda063884b5d2fda67b'
```

```
[37]: repo.log()
```

```
* a=fcd82f86e39b19c3e5351dda063884b5d2fda67b (testbranch) : mutated sample `0` of␣
↪`dummy_column` to new value
* a=eaee002ed9c6e949c3657bd50e3949d6a459d50e : first commit with a single sample␣
↪added to a dummy column
```

```
[38]: co.metadata['hello'] = 'world'
```

```
[39]: co.commit('added hellow world metadata')
```

```
[39]: 'a=69a08ca41ca1f5577fb0ffcf59d4d1585f614c4d'
```

```
[40]: co.close()
```

Looking at our history how, we see that none of the original branches reference our first commit anymore.

```
[41]: repo.log()
```

```
* a=69a08ca41ca1f5577fb0ffcf59d4d1585f614c4d (testbranch) : added hellow world␣
↪metadata
* a=fcd82f86e39b19c3e5351dda063884b5d2fda67b : mutated sample `0` of `dummy_column`␣
↪to new value
* a=eaee002ed9c6e949c3657bd50e3949d6a459d50e : first commit with a single sample␣
↪added to a dummy column
```

We can check the history of the `master` branch by specifying it as an argument to the `log()` method.

```
[42]: repo.log('master')
```

```
* a=c1cf1bd6863ed0b95239d2c9e1a6c6cc65569e94 (master) (new) : commit on `new` branch␣
↪adding a sample to dummy_arrayset
* a=eaee002ed9c6e949c3657bd50e3949d6a459d50e : first commit with a single sample␣
↪added to a dummy column
```

Merging (Part 2) Three Way Merge

If we now want to merge the changes on testbranch into master, we can't just follow a simple linear history; **the branches have diverged**.

For this case, Hangar implements a **Three Way Merge** algorithm which does the following: - Find the most recent common ancestor commit present in both the testbranch and master branches - Compute what changed between the common ancestor and each branch's HEAD commit - Check if any of the changes conflict with each other (more on this in a later tutorial) - If no conflicts are present, compute the results of the merge between the two sets of changes - Create a new commit containing the merge results reference both branch HEADs as parents of the new commit, and update the base branch HEAD to that new commit's commit_hash

```
[43]: co = repo.checkout(write=True, branch='master')
```

Once again, as a user, the details are completely irrelevant, and the operation occurs from the same one-liner call we used before for the FF Merge.

```
[44]: co.merge(message='merge of testbranch into master', dev_branch='testbranch')
```

```
Selected 3-Way Merge Strategy
```

```
[44]: 'a=002041fe8d8846b06f33842964904b627de55214'
```

If we now look at the log, we see that this has a much different look than before. The three way merge results in a history which references changes made in both diverged branches, and unifies them in a single commit

```
[45]: repo.log()
```

```
*   a=002041fe8d8846b06f33842964904b627de55214 (master) : merge of testbranch into␣
↪master
|\
| * a=69a08ca41ca1f5577fb0ffcf59d4d1585f614c4d (testbranch) : added hellow world␣
↪metadata
| * a=fcd82f86e39b19c3e5351dda063884b5d2fda67b : mutated sample `0` of `dummy_column`␣
↪to new value
* | a=c1cf1bd6863ed0b95239d2c9e1a6c6cc65569e94 (new) : commit on `new` branch adding␣
↪a sample to dummy_arrayset
|/
* a=eaee002ed9c6e949c3657bd50e3949d6a459d50e : first commit with a single sample␣
↪added to a dummy column
```

Manually inspecting the merge result to verify it matches our expectations

dummy_arrayset should contain two arrays, key 1 was set in the previous commit originally made in new and merged into master. Key 0 was mutated in testbranch and unchanged in master, so the update from testbranch is kept.

There should be one metadata sample with they key hello and the value "world".

```
[46]: co.columns
```

```
[46]: Hangar Columns
    Writeable         : True
    Number of Columns : 1
    Column Names / Partial Remote References:
      - dummy_column / False
```

```
[47]: co.columns['dummy_column']
```

```
[47]: Hangar FlatSampleWriter
          Column Name                : dummy_column
          Writeable                  : True
          Column Type                : ndarray
          Column Layout              : flat
          Schema Type                : fixed_shape
          DType                      : uint16
          Shape                      : (10,)
          Number of Samples          : 2
          Partial Remote Data Refs : False
```

```
[49]: co['dummy_column', ['0', '1']]
```

```
[49]: [array([50, 51, 52, 53, 54, 55, 56, 57, 58, 59], dtype=uint16),
       array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10], dtype=uint16)]
```

```
[50]: co.metadata
```

```
[50]: Hangar Metadata
          Writeable: True
          Number of Keys: 1
```

```
[51]: co.metadata['hello']
```

```
[51]: 'world'
```

The Merge was a success!

```
[52]: co.close()
```

Conflicts

Now that we've seen merging in action, the next step is to talk about conflicts.

How Are Conflicts Detected?

Any merge conflicts can be identified and addressed ahead of running a `merge` command by using the built in *diff* tools. When diffing commits, Hangar will provide a list of conflicts which it identifies. In general these fall into 4 categories:

1. **Additions** in both branches which created new keys (samples / columns / metadata) with non-compatible values. For samples & metadata, the hash of the data is compared, for columns, the schema specification is checked for compatibility in a method custom to the internal workings of Hangar.
2. **Removal** in `Master Commit/Branch` **& Mutation** in `Dev Commit / Branch`. Applies for samples, columns, and metadata identically.
3. **Mutation** in `Dev Commit/Branch` **& Removal** in `Master Commit / Branch`. Applies for samples, columns, and metadata identically.
4. **Mutations** on keys of both branches to non-compatible values. For samples & metadata, the hash of the data is compared; for columns, the schema specification is checked for compatibility in a method custom to the internal workings of Hangar.

Let's make a merge conflict

To force a conflict, we are going to checkout the new branch and set the metadata key hello to the value foo conflict... BOO!. Then if we try to merge this into the testbranch branch (which set hello to a value of world) we see how hangar will identify the conflict and halt without making any changes.

Automated conflict resolution will be introduced in a future version of Hangar, for now it is up to the user to manually resolve conflicts by making any necessary changes in each branch before reattempting a merge operation.

```
[53]: co = repo.checkout(write=True, branch='new')
```

```
[54]: co.metadata['hello'] = 'foo conflict... BOO!'
```

```
[55]: co.commit ('commit on new branch to hello metadata key so we can demonstrate a␣
      ↪conflict')
```

```
[55]: 'a=95896880b33fc06a3c2359a03408f07c87bcc8c0'
```

```
[56]: repo.log()
```

```
* a=95896880b33fc06a3c2359a03408f07c87bcc8c0 (new) : commit on new branch to hello␣
↪metadata key so we can demonstrate a conflict
* a=c1cf1bd6863ed0b95239d2c9e1a6c6cc65569e94 : commit on `new` branch adding a sample␣
↪to dummy_arrayset
* a=eaee002ed9c6e949c3657bd50e3949d6a459d50e : first commit with a single sample␣
↪added to a dummy column
```

When we attempt the merge, an exception is thrown telling us there is a conflict!

```
[57]: co.merge(message='this merge should not happen', dev_branch='testbranch')
```

```
Selected 3-Way Merge Strategy
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-57-1a98dce1852b> in <module>
----> 1 co.merge(message='this merge should not happen', dev_branch='testbranch')

~/projects/tensorwerk/hangar/hangar-py/src/hangar/checkout.py in merge(self, message,␣
↪dev_branch)
   1027                 dev_branch=dev_branch,
   1028                 repo_path=self._repo_path,
-> 1029                 writer_uuid=self._writer_lock)
   1030
   1031         for asetHandle in self._columns.values():

~/projects/tensorwerk/hangar/hangar-py/src/hangar/merger.py in select_merge_
↪algorithm(message, branchenv, stageenv, refenv, stagehashenv, master_branch, dev_
↪branch, repo_path, writer_uuid)
    136
    137     except ValueError as e:
--> 138         raise e from None
    139
    140     finally:

~/projects/tensorwerk/hangar/hangar-py/src/hangar/merger.py in select_merge_
↪algorithm(message, branchenv, stageenv, refenv, stagehashenv, master_branch, dev_
↪branch, repo_path, writer_uuid)
```

(continues on next page)

(continued from previous page)

```
    133                     refenv=refenv,
    134                     stagehashenv=stagehashenv,
--> 135                     repo_path=repo_path)
    136 
    137     except ValueError as e:

~/projects/tensorwerk/hangar/hangar-py/src/hangar/merger.py in _three_way_
↪merge(message, master_branch, masterHEAD, dev_branch, devHEAD, ancestorHEAD,␣
↪branchenv, stageenv, refenv, stagehashenv, repo_path)
    260         if conflict.conflict is True:
    261             msg = f'HANGAR VALUE ERROR:: Merge ABORTED with conflict:
↪{conflict}'
--> 262             raise ValueError(msg) from None
    263 
    264         with mEnv.begin(write=True) as txn:

ValueError: HANGAR VALUE ERROR:: Merge ABORTED with conflict: Conflicts(t1=[(b'l:hello
↪', b'2=d8fa6800caf496e637d965faac1a033e4636c2e6')], t21=[], t22=[], t3=[],␣
↪conflict=True)
```

Checking for Conflicts

Alternatively, use the diff methods on a checkout to test for conflicts before attempting a merge.

It is possible to diff between a checkout object and:

1. Another branch (*diff.branch()*)
2. A specified commit (*diff.commit()*)
3. Changes made in the staging area before a commit is made (*diff.staged()*) (for `write-enabled` checkouts only.)

Or via the *CLI status tool* between the staging area and any branch/commit (only a human readable summary is produced).

```
[58]: merge_results, conflicts_found = co.diff.branch('testbranch')
```

```
[59]: conflicts_found
```

```
[59]: Conflicts(t1=Changes(schema={}, samples=(), metadata=(MetadataRecordKey(key='hello'),
      ↪)), t21=Changes(schema={}, samples=(), metadata=()), t22=Changes(schema={},␣
      ↪samples=(), metadata=()), t3=Changes(schema={}, samples=(), metadata=()),␣
      ↪conflict=True)
```

```
[60]: conflicts_found.t1.metadata
```

```
[60]: (MetadataRecordKey(key='hello'),)
```

The type codes for a `Conflicts` `namedtuple` such as the one we saw:

```
Conflicts(t1=('hello',), t21=(), t22=(), t3=(), conflict=True)
```

are as follow:

- `t1`: Addition of key in master AND dev with different values.
- `t21`: Removed key in master, mutated value in dev.

- t22: Removed key in dev, mutated value in master.
- t3: Mutated key in both master AND dev to different values.
- conflict: Bool indicating if any type of conflict is present.

To resolve, remove the conflict

```
[61]: del co.metadata['hello']
      # resolved conflict by removing hello key
      co.commit('commit which removes conflicting metadata key')
```

```
[61]: 'a=e69ba8aeffc130c57d2ae0a8131c8ea59083cb62'
```

```
[62]: co.merge(message='this merge succeeds as it no longer has a conflict', dev_branch=
      ↪'testbranch')
```

```
Selected 3-Way Merge Strategy
```

```
[62]: 'a=ef7ddf4a4a216315d929bd905e78866e3ad6e4fd'
```

We can verify that history looks as we would expect via the log!

```
[63]: repo.log()
```

```
*   a=ef7ddf4a4a216315d929bd905e78866e3ad6e4fd (new) : this merge succeeds as it no␣
↪longer has a conflict
|\
* | a=e69ba8aeffc130c57d2ae0a8131c8ea59083cb62 : commit which removes conflicting␣
↪metadata key
* | a=95896880b33fc06a3c2359a03408f07c87bcc8c0 : commit on new branch to hello␣
↪metadata key so we can demonstrate a conflict
| * a=69a08ca41ca1f5577fb0ffcf59d4d1585f614c4d (testbranch) : added hellow world␣
↪metadata
| * a=fcd82f86e39b19c3e5351dda063884b5d2fda67b : mutated sample `0` of `dummy_column`␣
↪to new value
* | a=c1cf1bd6863ed0b95239d2c9e1a6c6cc65569e94 : commit on `new` branch adding a␣
↪sample to dummy_arrayset
|/
* a=eaee002ed9c6e949c3657bd50e3949d6a459d50e : first commit with a single sample␣
↪added to a dummy column
```

4.6.4 Part 3: Working With Remote Servers

This tutorial will introduce how to start a remote Hangar server, and how to work with *remotes* from the client side.

Particular attention is paid to the concept of a ***partially fetch* / *partial clone*** operations. This is a key component of the Hangar design which provides the ability to quickly and efficiently work with data contained in remote repositories whose full size would be significatly prohibitive to local use under most circumstances.

Note:

> At the time of writing, the API, user-facing functionality, client-server negotiation protocols, and test coverage of the remotes implementation is generally adqequate for this to serve as an "alpha" quality preview. However, please be warned that significantly less time has been spent in this module to optimize speed, refactor for simplicity, and assure stability under heavy loads than the rest of the Hangar core. While we can guarantee that your data is secure on disk, you may experience crashes from time to time when working with remotes. In addition, sending data over the wire should NOT be considered secure

> in ANY way. No in-transit encryption, user authentication, or secure access limitations are implemented at this moment. We realize the importance of these types of protections, and they are on our radar for the next release cycle. If you are interested in making a contribution to Hangar, this module contains a lot of low hanging fruit which would would provide drastic improvements and act as a good intro the the internal Hangar data model. Please get in touch with us to discuss!

Starting a Hangar Server

To start a Hangar server, navigate to the command line and simply execute:

```
$ hangar server
```

This will get a local server instance running at `localhost:50051`. The IP and port can be configured by setting the `--ip` and `--port` flags to the desired values in the command line.

A blocking process will begin in that terminal session. Leave it running while you experiment with connecting from a client repo.

Using Remotes with a Local Repository

The *CLI* is the easiest way to interact with the remote server from a local repository (though all functioanlity is mirrorred via the *repository API* (more on that later).

Before we begin we will set up a repository with some data, a few commits, two branches, and a merge.

Setup a Test Repo

As normal, we shall begin with creating a repository and adding some data. This should be familiar to you from previous tutorials.

[1]:

```
from hangar import Repository
import numpy as np
from tqdm import tqdm

testData = np.loadtxt('/Users/rick/projects/tensorwerk/hangar/dev/data/dota2Dataset/
↪dota2Test.csv', delimiter=',', dtype=np.uint8)
trainData = np.loadtxt('/Users/rick/projects/tensorwerk/hangar/dev/data/dota2Dataset/
↪dota2Train.csv', delimiter=',', dtype=np.uint16)

testName = 'test'
testPrototype = testData[0]
trainName = 'train'
trainPrototype = trainData[0]
```

[2]:

```
repo = Repository('/Users/rick/projects/tensorwerk/hangar/dev/intro/')
repo.init(user_name='Rick Izzo', user_email='rick@tensorwerk.com', remove_old=True)
co = repo.checkout(write=True)
```

```
Hangar Repo initialized at: /Users/rick/projects/tensorwerk/hangar/dev/intro/.hangar
```

```
/Users/rick/projects/tensorwerk/hangar/hangar-py/src/hangar/context.py:94:␣
↪UserWarning: No repository exists at /Users/rick/projects/tensorwerk/hangar/dev/
↪intro/.hangar, please use `repo.init()` method
  warnings.warn(msg, UserWarning)
```

[3]:

```
co.add_ndarray_column(testName, prototype=testPrototype)
testcol = co.columns[testName]

pbar = tqdm(total=testData.shape[0])
with testcol as tcol:
    for gameIdx, gameData in enumerate(testData):
        if (gameIdx % 500 == 0):
            pbar.update(500)
        tcol.append(gameData)
pbar.close()

co.commit('initial commit on master with test data')

repo.create_branch('add-train')
co.close()
repo.log()
```

```
10500it [00:02, 4286.17it/s]
```

```
* a=b98f6b65c0036489e53ddaf2b30bf797ddc40da0 (add-train) (master) : initial commit on␣
↪master with test data
```

[6]:

```
co = repo.checkout(write=True, branch='add-train')

co.add_ndarray_column(trainName, prototype=trainPrototype)
traincol = co.columns[trainName]

pbar = tqdm(total=trainData.shape[0])
with traincol as trcol:
    for gameIdx, gameData in enumerate(trainData):
        if (gameIdx % 500 == 0):
            pbar.update(500)
        trcol.append(gameData)
pbar.close()

co.commit('added training data on another branch')
co.close()
repo.log()
```

```
93000it [00:22, 4078.73it/s]
```

```
* a=957d20e4b921f41975591cc8ee51a4a6912cb919 (add-train) : added training data on␣
↪another branch
* a=b98f6b65c0036489e53ddaf2b30bf797ddc40da0 (master) : initial commit on master with␣
↪test data
```

[7]:

```
co = repo.checkout(write=True, branch='master')
co.metadata['earaea'] = 'eara'
co.commit('more changes here')
co.close()
repo.log()
```

```
* a=bb1b108ef17b7d7667a2ff396f257d82bad11e1d (master) : more changes here
* a=b98f6b65c0036489e53ddaf2b30bf797ddc40da0 : initial commit on master with test data
```

Pushing to a Remote

We will use the *API remote add()* method to add a remote, however, this can also be done with the *CLI command*:

```
$ hangar remote add origin localhost:50051
```

```
[8]: repo.remote.add('origin', 'localhost:50051')
```

```
[8]: RemoteInfo(name='origin', address='localhost:50051')
```

Pushing is as simple as running the *push()* method from the *API* or *CLI*:

```
$ hangar push origin master
```

Push the `master` branch:

```
[9]: repo.remote.push('origin', 'master')
```

```
counting objects: 100%|| 2/2 [00:00<00:00,  5.47it/s]
pushing schemas: 100%|| 1/1 [00:00<00:00, 133.74it/s]
pushing data:  97%|| 10001/10294 [00:01<00:00, 7676.23it/s]
pushing metadata: 100%|| 1/1 [00:00<00:00, 328.50it/s]
pushing commit refs: 100%|| 2/2 [00:00<00:00, 140.73it/s]
```

```
[9]: 'master'
```

Push the `add-train` branch:

```
[10]: repo.remote.push('origin', 'add-train')
```

```
counting objects: 100%|| 1/1 [00:01<00:00,  1.44s/it]
pushing schemas: 100%|| 1/1 [00:00<00:00, 126.05it/s]
pushing data:  99%|| 92001/92650 [00:12<00:00, 7107.60it/s]
pushing metadata: 0it [00:00, ?it/s]
pushing commit refs: 100%|| 1/1 [00:00<00:00, 17.05it/s]
```

```
[10]: 'add-train'
```

Details of the Negotiation Processs

> The following details are not necessary to use the system, but may be of interest to some readers

When we push data, **we perform a negotation with the server** which basically occurs like this:

- Hi, I would like to push this branch, do you have it?
- If yes, what is the latest commit you record on it?
  - Is that the same commit I'm trying to push? If yes, abort.
  - Is that a commit I don't have? If yes, someone else has updated that branch, abort.
- Here's the commit digests which are parents of my branches head, which commits are you missing?
- Ok great, I'm going to scan through each of those commits to find the data hashes they contain. Tell me which ones you are missing.
- Thanks, now I'll send you all of the data corresponding to those hashes. It might be a lot of data, so we'll handle this in batches so that if my connection cuts out, we can resume this later

- Now that you have the data, I'm going to send the actual commit references for you to store, this isn't that much information, but you'll be sure to verify that I'm not trying to pull any funny buisness and send you incorrect data.
- Now that you've received everything, and have verified it matches what I told you it is, go ahead and make those commits I've pushed `available` as the `HEAD` of the branch I just sent. It's some good work that others will want!

When we want to fetch updates to a branch, essentially the exact same thing happens in reverse. Instead of asking the server what it doesn't have, we ask it what it does have, and then request the stuff that we are missing!

Partial Fetching and Clones

Now we will introduce one of the most important and unique features of Hangar remotes: Partial fetch/clone of data!

*There is a very real problem with keeping the full history of data - **it's huge*!* The size of data can very easily exceeds what can fit on (most) contributors laptops or personal workstations. This section explains how Hangar can handle working with columns which are prohibitively large to download or store on a single machine.

As mentioned in High Performance From Simplicity, under the hood Hangar deals with "Data" and "Bookkeeping" completely separately. We've previously covered what exactly we mean by Data in How Hangar Thinks About Data, so we'll briefly cover the second major component of Hangar here. In short "Bookkeeping" describes everything about the repository. By everything, we do mean that the Bookkeeping records describe everything: all commits, parents, branches, columns, samples, data descriptors, schemas, commit message, etc. Though complete, these records are fairly small (tens of MB in size for decently sized repositories with decent history), and are highly compressed for fast transfer between a Hangar client/server.

A brief technical interlude

> There is one very important (and rather complex) property which gives Hangar Bookeeping massive power: existence of some data piece is always known to Hangar and stored immutably once committed. However, the access pattern, backend, and locating information for this data piece may (and over time, will) be unique in every hangar repository instance.
>
> Though the details of how this works is well beyond the scope of this document, the following example may provide some insight into the implications of this property:
>
> If you clone some Hangar repository, Bookeeping says that "some number of data pieces exist" and they should retrieved from the server. However, the bookeeping records transfered in a fetch / push / clone operation do not include information about where that piece of data existed on the client (or server) computer. Two synced repositories can use completly different backends to store the data, in completly different locations, and it does not matter - Hangar only guarantees that when collaborators ask for a data sample in some checkout, that they will be provided with identical arrays, not that they will come from the same place or be stored in the same way. Only when data is actually retrieved the "locating information" is set for that repository instance. Because Hangar makes no assumptions about how/where it should retrieve some piece of data, or even an assumption that it exists on the local machine, and because records are small and completely describe history, once a machine has the Bookkeeping, it can decide what data it actually wants to materialize on its local disk! These partial fetch / partial clone operations can materialize any desired data, whether it be for a few records at the head branch, for all data in a commit, or for the entire historical data. A future release will even include the ability to stream data directly to a Hangar checkout and materialize the data in memory without having to save it to disk at all!

More importantly: since Bookkeeping describes all history, merging can be performed between branches which may contain partial (or even no) actual data. Aka **you don't need data on disk to merge changes into it.** It's an odd concept which will be shown in this tutorial

Cloning a Remote Repo

```
$ hangar clone localhost:50051
```

[11]:

```
cloneRepo = Repository('/Users/rick/projects/tensorwerk/hangar/dev/dota-clone/')
```

```
/Users/rick/projects/tensorwerk/hangar/hangar-py/src/hangar/context.py:94:␣
↪UserWarning: No repository exists at /Users/rick/projects/tensorwerk/hangar/dev/
↪dota-clone/.hangar, please use `repo.init()` method
  warnings.warn(msg, UserWarning)
```

When we perform the initial clone, we will only receive the `master` branch by default.

[12]:

```
cloneRepo.clone('rick izzo', 'rick@tensorwerk.com', 'localhost:50051', remove_
↪old=True)
```

```
fetching commit data refs:   0%|          | 0/2 [00:00<?, ?it/s]
```

```
Hangar Repo initialized at: /Users/rick/projects/tensorwerk/hangar/dev/dota-clone/.
↪hangar
```

```
fetching commit data refs: 100%|| 2/2 [00:00<00:00,  5.73it/s]
fetching commit spec: 100%|| 2/2 [00:00<00:00, 273.30it/s]
```

```
Hard reset requested with writer_lock: 27634b20-3c5b-4ee0-aac3-b5ce6cb7daf0
```

[12]:

```
'master'
```

[13]:

```
cloneRepo.log()
```

```
* a=bb1b108ef17b7d7667a2ff396f257d82bad11e1d (master) (origin/master) : more changes␣
↪here
* a=b98f6b65c0036489e53ddaf2b30bf797ddc40da0 : initial commit on master with test data
```

[14]:

```
cloneRepo.list_branches()
```

[14]:

```
['master', 'origin/master']
```

To get the `add-train` branch, we *fetch* it from the remote:

[15]:

```
cloneRepo.remote.fetch('origin', 'add-train')
```

```
fetching commit data refs: 100%|| 1/1 [00:01<00:00,  1.51s/it]
fetching commit spec: 100%|| 1/1 [00:00<00:00, 35.85it/s]
```

[15]:

```
'origin/add-train'
```

[16]:

```
cloneRepo.list_branches()
```

[16]:

```
['master', 'origin/add-train', 'origin/master']
```

[17]:

```
cloneRepo.log(branch='origin/add-train')
```

```
* a=957d20e4b921f41975591cc8ee51a4a6912cb919 (origin/add-train) : added training data␣
↪on another branch
* a=b98f6b65c0036489e53ddaf2b30bf797ddc40da0 : initial commit on master with test data
```

We will create a local branch from the `origin/add-train` branch, just like in `Git`

[18]:

```
cloneRepo.create_branch('add-train', 'a=957d20e4b921f41975591cc8ee51a4a6912cb919')
```

```
[18]: BranchHead(name='add-train', digest='a=957d20e4b921f41975591cc8ee51a4a6912cb919')
```

```
[19]: cloneRepo.list_branches()
```

```
[19]: ['add-train', 'master', 'origin/add-train', 'origin/master']
```

```
[20]: cloneRepo.log(branch='add-train')
```

```
* a=957d20e4b921f41975591cc8ee51a4a6912cb919 (add-train) (origin/add-train) : added␣
↪training data on another branch
* a=b98f6b65c0036489e53ddaf2b30bf797ddc40da0 : initial commit on master with test data
```

Checking out a Parial Clone/Fetch

When we *fetch*/*clone*, the transfers are very quick, because only the commit records/history were retrieved. The data was not sent, because it may be very large to get the entire data across all of history.

When you check out a commit with partial data, you will be shown a warning indicating that some data is not available locally. An error is raised if you try to access that particular sample data. Otherwise, everything will appear as normal.

```
[21]: co = cloneRepo.checkout(branch='master')
```

```
 * Checking out BRANCH: master with current HEAD:␣
↪a=bb1b108ef17b7d7667a2ff396f257d82bad11e1d
```

```
/Users/rick/projects/tensorwerk/hangar/hangar-py/src/hangar/columns/constructors.py:
↪45: UserWarning: Column: test contains `reference-only` samples, with actual data␣
↪residing on a remote server. A `fetch-data` operation is required to access these␣
↪samples.
  f'operation is required to access these samples.', UserWarning)
```

```
[22]: co
```

```
[22]: Hangar ReaderCheckout
          Writer       : False
          Commit Hash  : a=bb1b108ef17b7d7667a2ff396f257d82bad11e1d
          Num Columns  : 1
          Num Metadata : 1
```

we can see from the `repr` that the columns contain partial remote references

```
[23]: co.columns
```

```
[23]: Hangar Columns
          Writeable         : False
          Number of Columns : 1
          Column Names / Partial Remote References:
            - test / True
```

```
[24]: co.columns['test']
```

```
[24]: Hangar FlatSampleReader
          Column Name              : test
          Writeable                : False
          Column Type              : ndarray
          Column Layout            : flat
```

(continues on next page)

(continued from previous page)

```
    Schema Type              : fixed_shape
    DType                    : uint8
    Shape                    : (117,)
    Number of Samples        : 10294
    Partial Remote Data Refs : True
```

[25]:
```
testKey = next(co.columns['test'].keys())
```

[26]:
```
co.columns['test'][testKey]
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-26-cb069e761eb3> in <module>
----> 1 co.columns['test'][testKey]

~/projects/tensorwerk/hangar/hangar-py/src/hangar/columns/layout_flat.py in __getitem_
↪_(self, key)
    222         """
    223         spec = self._samples[key]
--> 224         return self._be_fs[spec.backend].read_data(spec)
    225
    226     def get(self, key: KeyType, default=None):

~/projects/tensorwerk/hangar/hangar-py/src/hangar/backends/remote_50.py in read_
↪data(self, hashVal)
    172     def read_data(self, hashVal: REMOTE_50_DataHashSpec) -> None:
    173         raise FileNotFoundError(
--> 174             f'data hash spec: {REMOTE_50_DataHashSpec} does not exist on this␣
↪machine. '
    175             f'Perform a `data-fetch` operation to retrieve it from the remote␣
↪server.')
    176

FileNotFoundError: data hash spec: <class 'hangar.backends.specs.REMOTE_50_
↪DataHashSpec'> does not exist on this machine. Perform a `data-fetch` operation to␣
↪retrieve it from the remote server.
```

Fetching Data from a Remote

To retrieve the data, we use the *fetch_data()* method (accessible via the *API* or *fetch-data* via the CLI).

The amount / type of data to retrieve is extremly configurable via the following options:

`Remotes.`**`fetch_data`**(*remote: str, branch: str = None, commit: str = None, *, column_names: Optional[Sequence[str]] = None, max_num_bytes: int = None, retrieve_all_history: bool = False*) → List[str]

Retrieve the data for some commit which exists in a *partial* state.

Parameters

- **remote** (*str*) – name of the remote to pull the data from
- **branch** (*str, optional*) – The name of a branch whose HEAD will be used as the data fetch point. If None, `commit` argument expected, by default None

- **commit** (*str, optional*) – Commit hash to retrieve data for, If None, `branch` argument expected, by default None
- **column_names** (*Optional[Sequence[str]]*) – Names of the columns which should be retrieved for the particular commits, any columns not named will not have their data fetched from the server. Default behavior is to retrieve all columns
- **max_num_bytes** (*Optional[int]*) – If you wish to limit the amount of data sent to the local machine, set a *max_num_bytes* parameter. This will retrieve only this amount of data from the server to be placed on the local disk. Default is to retrieve all data regardless of how large.
- **retrieve_all_history** (*Optional[bool]*) – if data should be retrieved for all history accessible by the parents of this commit HEAD. by default False

Returns commit hashes of the data which was returned.

Return type List[str]

Raises

- **ValueError** – if branch and commit args are set simultaneously.
- **ValueError** – if specified commit does not exist in the repository.
- **ValueError** – if branch name does not exist in the repository.

This will retrieve all the data on the `master` branch, but not on the `add-train` branch.

```
[29]: cloneRepo.remote.fetch_data('origin', branch='master')
```

```
counting objects: 100%|| 1/1 [00:00<00:00, 27.45it/s]
fetching data: 100%|| 10294/10294 [00:01<00:00, 6664.60it/s]
```

```
[29]: ['a=bb1b108ef17b7d7667a2ff396f257d82bad11e1d']
```

```
[30]: co = cloneRepo.checkout(branch='master')
```

```
 * Checking out BRANCH: master with current HEAD:␣
↪a=bb1b108ef17b7d7667a2ff396f257d82bad11e1d
```

```
[31]: co
```

```
[31]: Hangar ReaderCheckout
          Writer       : False
          Commit Hash  : a=bb1b108ef17b7d7667a2ff396f257d82bad11e1d
          Num Columns  : 1
          Num Metadata : 1
```

Unlike before, we see that there is no partial references from the `repr`

```
[32]: co.columns
```

```
[32]: Hangar Columns
          Writeable         : False
          Number of Columns : 1
          Column Names / Partial Remote References:
            - test / False
```

```
[33]: co.columns['test']
```

```
[33]: Hangar FlatSampleReader
          Column Name              : test
          Writeable                : False
          Column Type              : ndarray
          Column Layout            : flat
          Schema Type              : fixed_shape
          DType                    : uint8
          Shape                    : (117,)
          Number of Samples        : 10294
          Partial Remote Data Refs : False
```

When we access the data this time, it is available and retrieved as requested!

```
[34]: co['test', testKey]
```

```
[34]: array([255, 223,   8,   2,   0, 255,   0,   0,   0,   0,   0,   0,   1,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   1,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   1,   0,   0,   0, 255,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   1,   0,   0,   0, 255,   0,   0,
               0,   0,   0,   0,   0, 255,   0,   0,   0,   0,   1,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
               0,   0,   0, 255,   0,   0,   0,   0,   0,   0,   0,   0,   0],
             dtype=uint8)
```

```
[35]: co.close()
```

Working with mixed local / remote checkout Data

If we were to checkout the `add-train` branch now, we would see that there is no `arrayset "train"` data, but there will be data common to the ancestor that `master` and `add-train` share.

```
[36]: cloneRepo.log('add-train')
```

```
* a=957d20e4b921f41975591cc8ee51a4a6912cb919 (add-train) (origin/add-train) : added␣
↪training data on another branch
* a=b98f6b65c0036489e53ddaf2b30bf797ddc40da0 : initial commit on master with test data
```

In this case, the common ancestor is commit: `9b93b393e8852a1fa57f0170f54b30c2c0c7d90f`

To show that there is no data on the `add-train` branch:

```
[37]: co = cloneRepo.checkout(branch='add-train')
```

```
 * Checking out BRANCH: add-train with current HEAD:␣
↪a=957d20e4b921f41975591cc8ee51a4a6912cb919
```

```
/Users/rick/projects/tensorwerk/hangar/hangar-py/src/hangar/columns/constructors.py:
↪45: UserWarning: Column: train contains `reference-only` samples, with actual data␣
↪residing on a remote server. A `fetch-data` operation is required to access these␣
↪samples.
  f'operation is required to access these samples.', UserWarning)
```

```
[38]: co
```

[38]:
```
Hangar ReaderCheckout
    Writer       : False
    Commit Hash  : a=957d20e4b921f41975591cc8ee51a4a6912cb919
    Num Columns  : 2
    Num Metadata : 0
```

[39]:
```
co.columns
```

[39]:
```
Hangar Columns
    Writeable         : False
    Number of Columns : 2
    Column Names / Partial Remote References:
      - test / False
      - train / True
```

[40]:
```
co['test', testKey]
```

[40]:
```
array([255, 223,   8,   2,   0, 255,   0,   0,   0,   0,   0,   0,   1,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   1,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   1,   0,   0,   0, 255,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   1,   0,   0,   0, 255,   0,   0,
         0,   0,   0,   0,   0, 255,   0,   0,   0,   0,   1,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0, 255,   0,   0,   0,   0,   0,   0,   0,   0,   0],
      dtype=uint8)
```

[41]:
```
trainKey = next(co.columns['train'].keys())
```

[42]:
```
co.columns['train'][trainKey]
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-42-549d3e1dc7a1> in <module>
----> 1 co.columns['train'][trainKey]

~/projects/tensorwerk/hangar/hangar-py/src/hangar/columns/layout_flat.py in __getitem_
 ↪_(self, key)
    222         """
    223         spec = self._samples[key]
--> 224         return self._be_fs[spec.backend].read_data(spec)
    225
    226     def get(self, key: KeyType, default=None):

~/projects/tensorwerk/hangar/hangar-py/src/hangar/backends/remote_50.py in read_
 ↪data(self, hashVal)
    172     def read_data(self, hashVal: REMOTE_50_DataHashSpec) -> None:
    173         raise FileNotFoundError(
--> 174             f'data hash spec: {REMOTE_50_DataHashSpec} does not exist on this␣
 ↪machine. '
    175             f'Perform a `data-fetch` operation to retrieve it from the remote␣
 ↪server.')
    176

FileNotFoundError: data hash spec: <class 'hangar.backends.specs.REMOTE_50_
 ↪DataHashSpec'> does not exist on this machine. Perform a `data-fetch` operation to␣
 ↪retrieve it from the remote server.
```

(continues on next page)

(continued from previous page)

```
[43]: co.close()
```

Merging Branches with Partial Data

Even though we don't have the actual data references in the `add-train` branch, it is still possible to merge the two branches!

This is possible because Hangar doesn't use the data contents in its internal model of checkouts / commits, but instead thinks of a checkouts as a sequence of columns / metadata / keys & their associated data hashes (which are very small text records; ie. "bookkeeping"). To show this in action, lets merge the two branches `master` (containing all data locally) and `add-train` (containing partial remote references for the `train` arrayset) together and push it to the Remote!

```
[44]: cloneRepo.log('master')
```

```
* a=bb1b108ef17b7d7667a2ff396f257d82bad11e1d (master) (origin/master) : more changes␣
↪here
* a=b98f6b65c0036489e53ddaf2b30bf797ddc40da0 : initial commit on master with test data
```

```
[45]: cloneRepo.log('add-train')
```

```
* a=957d20e4b921f41975591cc8ee51a4a6912cb919 (add-train) (origin/add-train) : added␣
↪training data on another branch
* a=b98f6b65c0036489e53ddaf2b30bf797ddc40da0 : initial commit on master with test data
```

Perform the Merge

```
[46]: cloneRepo.merge('merge commit here', 'master', 'add-train')
```

```
Selected 3-Way Merge Strategy
```

```
[46]: 'a=ace3dacbd94f475664ee136dcf05430a2895aca3'
```

IT WORKED!

```
[47]: cloneRepo.log()
```

```
*   a=ace3dacbd94f475664ee136dcf05430a2895aca3 (master) : merge commit here
|\
* | a=bb1b108ef17b7d7667a2ff396f257d82bad11e1d (origin/master) : more changes here
| * a=957d20e4b921f41975591cc8ee51a4a6912cb919 (add-train) (origin/add-train) : added␣
↪training data on another branch
|/
* a=b98f6b65c0036489e53ddaf2b30bf797ddc40da0 : initial commit on master with test data
```

We can check the summary of the master commit to check that the contents are what we expect (containing both `test` and `train` columns)

```
[48]: cloneRepo.summary()
```

```
Summary of Contents Contained in Data Repository

==================
| Repository Info
|-----------------
```

(continues on next page)

(continued from previous page)

```
|  Base Directory: /Users/rick/projects/tensorwerk/hangar/dev/dota-clone
|  Disk Usage: 42.03 MB

===================
| Commit Details
-------------------
|  Commit: a=ace3dacbd94f475664ee136dcf05430a2895aca3
|  Created: Tue Feb 25 19:18:30 2020
|  By: rick izzo
|  Email: rick@tensorwerk.com
|  Message: merge commit here

==================
| DataSets
|-----------------
|  Number of Named Columns: 2
|
|  * Column Name: ColumnSchemaKey(column="test", layout="flat")
|    Num Data Pieces: 10294
|    Details:
|    - column_layout: flat
|    - column_type: ndarray
|    - schema_type: fixed_shape
|    - shape: (117,)
|    - dtype: uint8
|    - backend: 10
|    - backend_options: {}
|
|  * Column Name: ColumnSchemaKey(column="train", layout="flat")
|    Num Data Pieces: 92650
|    Details:
|    - column_layout: flat
|    - column_type: ndarray
|    - schema_type: fixed_shape
|    - shape: (117,)
|    - dtype: uint16
|    - backend: 10
|    - backend_options: {}

==================
| Metadata:
|-----------------
|  Number of Keys: 1
```

Pushing the Merge back to the Remote

To push this merge back to our original copy of the Repository (`repo`), we just push the `master` branch back to the remote via the API or CLI.

[49]:
```
cloneRepo.remote.push('origin', 'master')
```

```
counting objects: 100%|| 1/1 [00:00<00:00,  1.02it/s]
pushing schemas: 0it [00:00, ?it/s]
pushing data: 0it [00:00, ?it/s]
```

(continues on next page)

(continued from previous page)

```
pushing metadata: 0it [00:00, ?it/s]
pushing commit refs: 100%|| 1/1 [00:00<00:00, 34.26it/s]
```

[49]:
```
'master'
```

Looking at our current state of our other instance of the repo `repo` we see that the merge changes aren't yet propogated to it (since it hasn't fetched from the remote yet).

[50]:
```
repo.log()
```

```
* a=bb1b108ef17b7d7667a2ff396f257d82bad11e1d (master) (origin/master) : more changes␣
↪here
* a=b98f6b65c0036489e53ddaf2b30bf797ddc40da0 : initial commit on master with test data
```

To fetch the merged changes, just *fetch()* the branch as normal. Like all fetches, this will be a fast operation, as it will be a `partial fetch` operation, not actually transfering the data.

[51]:
```
repo.remote.fetch('origin', 'master')
```

```
fetching commit data refs: 100%|| 1/1 [00:01<00:00,  1.33s/it]
fetching commit spec: 100%|| 1/1 [00:00<00:00, 37.61it/s]
```

[51]:
```
'origin/master'
```

[52]:
```
repo.log('origin/master')
```

```
*   a=ace3dacbd94f475664ee136dcf05430a2895aca3 (origin/master) : merge commit here
|\
* | a=bb1b108ef17b7d7667a2ff396f257d82bad11e1d (master) : more changes here
| * a=957d20e4b921f41975591cc8ee51a4a6912cb919 (add-train) (origin/add-train) : added␣
↪training data on another branch
|/
* a=b98f6b65c0036489e53ddaf2b30bf797ddc40da0 : initial commit on master with test data
```

To bring our `master` branch up to date is a simple fast-forward merge.

[53]:
```
repo.merge('ff-merge', 'master', 'origin/master')
```

```
Selected Fast-Forward Merge Strategy
```

[53]:
```
'a=ace3dacbd94f475664ee136dcf05430a2895aca3'
```

[54]:
```
repo.log()
```

```
*   a=ace3dacbd94f475664ee136dcf05430a2895aca3 (master) (origin/master) : merge␣
↪commit here
|\
* | a=bb1b108ef17b7d7667a2ff396f257d82bad11e1d : more changes here
| * a=957d20e4b921f41975591cc8ee51a4a6912cb919 (add-train) (origin/add-train) : added␣
↪training data on another branch
|/
* a=b98f6b65c0036489e53ddaf2b30bf797ddc40da0 : initial commit on master with test data
```

Everything is as it should be! Now, try it out for yourself!

[55]:
```
repo.summary()
```

```
Summary of Contents Contained in Data Repository
```

(continues on next page)

(continued from previous page)

```
==================
| Repository Info
|-----------------
|  Base Directory: /Users/rick/projects/tensorwerk/hangar/dev/intro
|  Disk Usage: 77.43 MB

===================
| Commit Details
-------------------
|  Commit: a=ace3dacbd94f475664ee136dcf05430a2895aca3
|  Created: Tue Feb 25 19:18:30 2020
|  By: rick izzo
|  Email: rick@tensorwerk.com
|  Message: merge commit here

==================
| DataSets
|-----------------
|  Number of Named Columns: 2
|
|  * Column Name: ColumnSchemaKey(column="test", layout="flat")
|    Num Data Pieces: 10294
|    Details:
|    - column_layout: flat
|    - column_type: ndarray
|    - schema_type: fixed_shape
|    - shape: (117,)
|    - dtype: uint8
|    - backend: 10
|    - backend_options: {}
|
|  * Column Name: ColumnSchemaKey(column="train", layout="flat")
|    Num Data Pieces: 92650
|    Details:
|    - column_layout: flat
|    - column_type: ndarray
|    - schema_type: fixed_shape
|    - shape: (117,)
|    - dtype: uint16
|    - backend: 10
|    - backend_options: {}

==================
| Metadata:
|-----------------
|  Number of Keys: 1
```

[ ]:

4.6.5 Dataloaders for Machine Learning (Tensorflow & PyTorch)

This tutorial acts as a step by step guide for fetching, preprocessing, storing and loading the MS-COCO dataset for image captioning using deep learning. We have chosen **image captioning** for this tutorial not by accident. For such an application, the dataset required will have both fixed shape (image) and variably shaped (caption because it's sequence of natural language) data. This diversity should help the user to get a mental model about how flexible and easy is to plug Hangar to the existing workflow.

You will use the MS-COCO dataset to train our model. The dataset contains over 82,000 images, each of which has at least 5 different caption annotations.

This tutorial assumes you have downloaded and extracted the MS-COCO dataset in the current directory. If you haven't yet, shell commands below should help you do it (beware, it's about 14 GB data). If you are on Windows, please find the equivalent commands to get the dataset downloaded.

```
wget http://images.cocodataset.org/zips/train2014.zip
unzip train2014.zip
rm train2014.zip
wget http://images.cocodataset.org/annotations/annotations_trainval2014.zip
unzip annotations_trainval2014.zip
rm annotations_trainval2014.zip
```

Let's install the required packages in our environment. We will be using Tensorflow 1.14 in this tutorial but it should work in all the Tensorflow versions starting from 1.12. But do let us know if you face any hiccups. Install below-given packages before continue. Apart from Tensorflow and Hangar, we use SpaCy for pre-processing the captions. SpaCy is probably the most widely used natural language toolkit now.

```
tensorflow==1.14.0
hangar
spacy==2.1.8
```

One more thing before jumping into the tutorial: we need to download the SpaCy English model `en_core_web_md` which cannot be dynamically loaded. Which means that it must be downloaded with the below command outside this runtime and should reload this runtime.

```
python -m spacy download en_core_web_md
```

Once all the dependencies are installed and loaded, we can start building our hangar repository.

Hangar Repository creation and column init

We will create a repository and initialize one column named `images` now for a quick demo of how Tensorflow dataloader work. Then we wipe the current repository and create new columns for later portions.

[ ]:

```
repo_path = 'hangar_repo'
username = 'hhsecond'
email = 'sherin@tensorwerk.com'
img_shape = (299, 299, 3)
image_dir = '/content/drive/My Drive/train2014'
annotation_file = ''
import logging
logging.getLogger("tensorflow").setLevel(logging.ERROR)
```

[2]:

```
import os
from hangar import Repository
import tensorflow as tf
```

(continues on next page)

(continued from previous page)

```
import numpy as np

tf.compat.v1.enable_eager_execution()


if not os.path.isdir(repo_path):
    os.mkdir(repo_path)

repo = Repository(repo_path)
repo.init(user_name=username, user_email=email, remove_old=True)
co = repo.checkout(write=True)

images_column = co.add_ndarray_column('images', shape=img_shape, dtype=np.uint8,)
co.commit('column init')
co.close()
```

```
Hangar Repo initialized at: hangar_repo/.hangar
```

Add sample images

Here we add few images to the repository and show how we can load this data as Tensorflow dataloader. We use the idea we learn here in the later portions to build a fully fledged training loop.

[ ]:

```
import os
from PIL import Image


co = repo.checkout(write=True)
images_column = co.columns['images']
try:
    for i, file in enumerate(os.listdir(image_dir)):
        pil_img = Image.open(os.path.join(image_dir, file))
        if pil_img.mode == 'L':
            pil_img = pil_img.convert('RGB')
        img = pil_img.resize(img_shape[:-1])
        img = np.array(img)
        images_column[i] = img
        if i != 0 and i % 2 == 0:  # stopping at 2th image
            break
except Exception as e:
    print('Exception', e)
    co.close()
    raise e
co.commit('added image')
co.close()
```

Let's make a Tensorflow dataloader

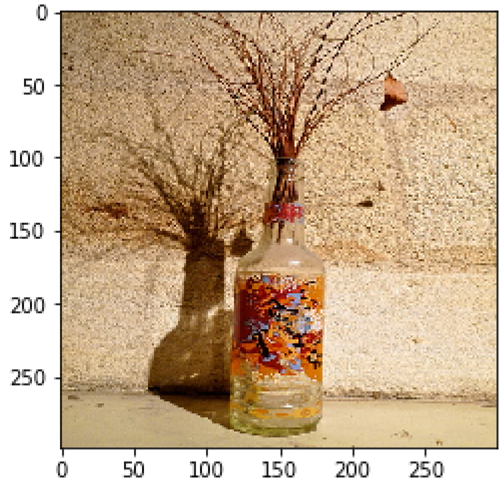
Hangar provides `make_tf_dataset` & `make_torch_dataset` for creating Tensorflow & PyTorch datasets from Hangar columns. You can read more about it in the documentation. Next we'll make a Tensorflow dataset and loop over it to make sure we have got a proper Tensorflow dataset.

```
[ ]: from hangar import make_tf_dataset
```

```
[5]: from matplotlib.pyplot import imshow
     co = repo.checkout()
     image_column = co.columns['images']
     dataset = make_tf_dataset(image_column)
     for image in dataset:
         imshow(image[0].numpy())
         break
```

```
 * Checking out BRANCH: master with current HEAD:␣
↪b769f6d49a7dbb3dcd4f7c6e1c2a32696fd4128f
<class 'hangar.columns.arrayset.ArraysetDataReader'>(repo_pth=hangar_repo/.hangar,␣
↪aset_name=images, default_schema_hash=b6edf0320f20, isVar=False, varMaxShape=(299,␣
↪299, 3), varDtypeNum=2, mode=r)
```

```
/usr/local/lib/python3.6/dist-packages/hangar/dataloaders/tfloader.py:88: UserWarning:
↪ Dataloaders are experimental in the current release.
  warnings.warn("Dataloaders are experimental in the current release.", UserWarning)
```



New columns

For our example, we would need two columns. One for the image and another one for captions. Let's wipe our existing repository (`remove_old` argument in `repo.init` does this) and create these columns

```
[6]: repo = Repository(repo_path)
     repo.init(user_name=username, user_email=email, remove_old=True)
     co = repo.checkout(write=True)

     images_column = co.add_ndarray_column('images', shape=img_shape, dtype=np.uint8)
     captions_column = co.add_ndarray_column('captions', shape=(60,), dtype=np.float,␣
     ↪variable_shape=True)
     co.commit('column init')
```

(continues on next page)

(continued from previous page)

```
co.close()
```

```
Hangar Repo initialized at: hangar_repo/.hangar
```

Store image and captions to Hangar repo

Each image will be converted to RGB channels with dtype `uint8`. Each caption will be prepended with `START` token and ended with `END` token before converting them to floats. We have another preprocessing stage for images later.

We'll start with loading the caption file:

[ ]:
```
import json
annotation_file = 'annotations/captions_train2014.json'
with open(annotation_file, 'r') as f:
    annotations = json.load(f)
```

[ ]:
```
import spacy
# if you have installed spacy and the model in the same notebook session, you might␣
↪need to restart the runtime to get it into the scope
nlp = spacy.load('en_core_web_md')
```

[ ]:
```
def sent2index(sent):
    """
    Convert sentence to an array of indices using SpaCy
    """
    ids = []
    doc = nlp(sent)
    for token in doc:
        if token.has_vector:
            id = nlp.vocab.vectors.key2row[token.norm]
        else:
            id = sent2index('UNK')[0]
        ids.append(id)
    return ids
```

Save the data to Hangar

[10]:
```
import os
from tqdm import tqdm

all_captions = []
all_img_name_vector = []
limit = 100  # if you are not planning to save the whole dataset to Hangar. Zero␣
↪means whole dataset

co = repo.checkout(write=True)
images_column = co.columns['images']
captions_column = co.columns['captions']
all_files = set(os.listdir(image_dir))
i = 0
with images_column, captions_column:
    for annot in tqdm(annotations['annotations']):
        if limit and i > limit:
```

(continues on next page)

(continued from previous page)

```
            continue
        image_id = annot['image_id']
        assumed_image_paths = 'COCO_train2014_' + '%012d.jpg' % (image_id)
        if assumed_image_paths not in all_files:
            continue
        img_path = os.path.join(image_dir, assumed_image_paths)
        img = Image.open(img_path)
        if img.mode == 'L':
            img = img.convert('RGB')
        img = img.resize(img_shape[:-1])
        img = np.array(img)
        cap = sent2index('sos ' + annot['caption'] + ' eos')
        cap = np.array(cap, dtype=np.float)
        key = images_column.append(img)
        captions_column[key] = cap
        if i % 1000 == 0 and i != 0:
            if co.diff.status() == 'DIRTY':
                co.commit(f'Added batch {i}')
        i += 1
co.commit('Added full data')
co.close()
```

```
100%|| 414113/414113 [00:03<00:00, 122039.19it/s]
```

Preprocess Images

Our image captioning network requires a pre-processed input. We use transfer learning for this with a pretrained InceptionV3 network which is available in Keras. But we have a problem. Preprocessing is costly and we don't want to do it all the time. Since Hangar is flexible enough to create multiple columns and let you call the group of column as a `dataset`, it is quite easy to do make a new column for the processed image and we don't have to do the preprocessing online but keep a preprocessed image in the new column in the same repository with the same key. Which means, we have three columns in our repository (all three has different samples with the same name): - `images` - `captions` - `processed_images`

Although we need only the `processed_images` for the network, we still keep the bare image in the repository in case we need to look into it later or if we decided to do some other preprocessing instead of InceptionV3 (it is always advised to keep the source truth with you).

[ ]:

```
import tensorflow as tf
tf.compat.v1.enable_eager_execution()
image_model = tf.keras.applications.InceptionV3(include_top=False, weights='imagenet')
new_input = image_model.input
hidden_layer = image_model.layers[-1].output
image_features_extract_model = tf.keras.Model(new_input, hidden_layer)


def process_image(img):
    img = tf.keras.applications.inception_v3.preprocess_input(img)
    img = np.expand_dims(img, axis=0)
    img = image_features_extract_model(img)
    return tf.reshape(img, (-1, img.shape[3]))
```

[ ]:

```
from hangar import Repository
import numpy as np
```

(continues on next page)

(continued from previous page)

```
repo_path = 'hangar_repo'

repo = Repository(repo_path)
co = repo.checkout(write=True)
images = co.columns['images']
sample_name = list(images.keys())[0]
prototype = process_image(images[sample_name]).numpy()
pimages = co.add_ndarray_column('processed_images', prototype=prototype)
```

Saving the pre-processed images to the new column

[6]:

```
from tqdm import tqdm

with pimages:
    for key in tqdm(images):
        pimages[key] = process_image(images[key]).numpy()

co.commit('processed image saved')
co.close()
```

```
100%|| 101/101 [00:11<00:00,  8.44it/s]
```

Dataloaders for training

We are using Tensorflow to build the network but how do we load this data from Hangar repository to Tensorflow?

A naive option would be to run through the samples and load the numpy arrays and pass that to the `sess.run` of Tensorflow. But that would be quite inefficient. Tensorflow uses multiple threads to load the data in memory and its dataloaders can prefetch the data before-hand so that your training loop doesn't get blocked while loading the data. Also, Tensoflow dataloaders brings batching, shuffling, etc. to the table prebuilt. That's cool but how to load data from Hangar to Tensorflow using TF dataset? Well, we have `make_tf_dataset` which accepts the list of columns as a parameter and returns a TF dataset object.

[7]:

```
from hangar import make_tf_dataset
co = repo.checkout()   # we don't need write checkout here
```

```
 * Checking out BRANCH: master with current HEAD:␣
↪3cbb3fbe7eb0e056ff97e75f41d26303916ef686
```

[8]:

```
BATCH_SIZE = 1
EPOCHS = 2
embedding_dim = 256
units = 512
vocab_size = len(nlp.vocab.vectors.key2row)
num_steps = 50


captions_dset = co.columns['captions']
pimages_dset = co.columns['processed_images']

dataset = make_tf_dataset([pimages_dset, captions_dset], shuffle=True)
```

```
<class 'hangar.columns.arrayset.ArraysetDataReader'>(repo_pth=hangar_repo/.hangar,␣
↪aset_name=processed_images, default_schema_hash=f230548212ab, isVar=False,␣
↪varMaxShape=(64, 2048), varDtypeNum=11, mode=r)
<class 'hangar.columns.arrayset.ArraysetDataReader'>(repo_pth=hangar_repo/.hangar,␣
↪aset_name=captions, default_schema_hash=4d60751421d5, isVar=True, varMaxShape=(60,),
↪ varDtypeNum=12, mode=r)
```

```
/usr/local/lib/python3.6/dist-packages/hangar/dataloaders/tfloader.py:88: UserWarning:
↪ Dataloaders are experimental in the current release.
  warnings.warn("Dataloaders are experimental in the current release.", UserWarning)
```

Padded Batching

Batching needs a bit more explanation here since the dataset does not just consist of fixed shaped data. We have two dataset in which one is for captions. As you know captions are sequences which can be variably shaped. So instead of using `dataset.batch` we need to use `dataset.padded_batch` which takes care of padding the tensors with the longest value in each dimension for each batch. This `padded_batch` needs the shape by which the user needs the batch to be padded. Unless you need customization, you can use the shape stored in the `dataset` object by `make_tf_dataset` function.

[9]:
```
output_shapes = tf.compat.v1.data.get_output_shapes(dataset)
output_shapes
```

[9]:
```
(TensorShape([Dimension(64), Dimension(2048)]), TensorShape([Dimension(None)]))
```

[ ]:
```
dataset = dataset.padded_batch(BATCH_SIZE, padded_shapes=output_shapes)
```

Build the network

Since we have the dataloaders ready, we can now build the network for image captioning and start training. Rest of this tutorial is a copy of an official Tensorflow tutorial which is available at https://tensorflow.org/beta/tutorials/text/image_captioning. The content of Tensorflow tutorial page is licensed under the Creative Commons Attribution 4.0 License, and code samples are licensed under the Apache 2.0 License. Access date: Aug 20 2019

In this example, you extract the features from the lower convolutional layer of InceptionV3 giving us a vector of shape (8, 8, 2048) and quash that to a shape of (64, 2048). We have stored the result of this already to our Hangar repo. This vector is then passed through the CNN Encoder (which consists of a single Fully connected layer). The RNN (here GRU) attends over the image to predict the next word.

[ ]:
```
class BahdanauAttention(tf.keras.Model):
    def __init__(self, units):
        super(BahdanauAttention, self).__init__()
        self.W1 = tf.keras.layers.Dense(units)
        self.W2 = tf.keras.layers.Dense(units)
        self.V = tf.keras.layers.Dense(1)

    def call(self, features, hidden):
        # features(CNN_encoder output) shape == (batch_size, 64, embedding_dim)
        # hidden shape == (batch_size, hidden_size)
        # hidden_with_time_axis shape == (batch_size, 1, hidden_size)
        hidden_with_time_axis = tf.expand_dims(hidden, 1)
        # score shape == (batch_size, 64, hidden_size)
        score = tf.nn.tanh(self.W1(features) + self.W2(hidden_with_time_axis))
        # attention_weights shape == (batch_size, 64, 1)
```

(continues on next page)

(continued from previous page)

```
        # you get 1 at the last axis because you are applying score to self.V
        attention_weights = tf.nn.softmax(self.V(score), axis=1)
        # context_vector shape after sum == (batch_size, hidden_size)
        context_vector = attention_weights * features
        context_vector = tf.reduce_sum(context_vector, axis=1)

        return context_vector, attention_weights
```

[ ]:
```
class CNN_Encoder(tf.keras.Model):
    # Since you have already extracted the features and dumped it using pickle
    # This encoder passes those features through a Fully connected layer
    def __init__(self, embedding_dim):
        super(CNN_Encoder, self).__init__()
        # shape after fc == (batch_size, 64, embedding_dim)
        self.fc = tf.keras.layers.Dense(embedding_dim)

    def call(self, x):
        x = self.fc(x)
        x = tf.nn.relu(x)
        return x
```

[ ]:
```
class RNN_Decoder(tf.keras.Model):
    def __init__(self, embedding_dim, units, vocab_size):
        super(RNN_Decoder, self).__init__()
        self.units = units
        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
        self.gru = tf.keras.layers.GRU(self.units,
                                       return_sequences=True,
                                       return_state=True,
                                       recurrent_initializer='glorot_uniform')
        self.fc1 = tf.keras.layers.Dense(self.units)
        self.fc2 = tf.keras.layers.Dense(vocab_size)
        self.attention = BahdanauAttention(self.units)

    def call(self, x, features, hidden):
        # defining attention as a separate model
        context_vector, attention_weights = self.attention(features, hidden)
        # x shape after passing through embedding == (batch_size, 1, embedding_dim)
        x = self.embedding(x)
        # x shape after concatenation == (batch_size, 1, embedding_dim + hidden_size)
        x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)
        # passing the concatenated vector to the GRU
        output, state = self.gru(x)
        # shape == (batch_size, max_length, hidden_size)
        x = self.fc1(output)
        # x shape == (batch_size * max_length, hidden_size)
        x = tf.reshape(x, (-1, x.shape[2]))
        # output shape == (batch_size * max_length, vocab)
        x = self.fc2(x)
        return x, state, attention_weights

    def reset_state(self, batch_size):
        return tf.zeros((batch_size, self.units))
```

[ ]:
```
def loss_function(real, pred):
```

(continues on next page)

(continued from previous page)

```
    mask = tf.math.logical_not(tf.math.equal(real, 0))
    loss_ = loss_object(real, pred)
    mask = tf.cast(mask, dtype=loss_.dtype)
    loss_ *= mask
    return tf.reduce_mean(loss_)
```

[ ]:

```
@tf.function
def train_step(img_tensor, target):
    loss = 0
    # initializing the hidden state for each batch
    # because the captions are not related from image to image
    hidden = decoder.reset_state(batch_size=target.shape[0])
    # TODO: do this dynamically: '<start>' == 2
    dec_input = tf.expand_dims([2] * BATCH_SIZE, 1)

    with tf.GradientTape() as tape:
        features = encoder(img_tensor)
        for i in range(1, target.shape[1]):
            # passing the features through the decoder
            predictions, hidden, _ = decoder(dec_input, features, hidden)
            loss += loss_function(target[:, i], predictions)
            # using teacher forcing
            dec_input = tf.expand_dims(target[:, i], 1)
    total_loss = (loss / int(target.shape[1]))
    trainable_variables = encoder.trainable_variables + decoder.trainable_variables

    gradients = tape.gradient(loss, trainable_variables)
    optimizer.apply_gradients(zip(gradients, trainable_variables))
    return loss, total_loss
```

[ ]:

```
encoder = CNN_Encoder(embedding_dim)
decoder = RNN_Decoder(embedding_dim, units, vocab_size)
optimizer = tf.keras.optimizers.Adam()
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True,␣
↪reduction='none')
```

Training

Here we consume the dataset we have made before by looping over it. The dataset returns the image tensor and target tensor (captions) which we will pass to `train_step` for training the network.

The encoder output, hidden state (initialized to 0) and the decoder input (which is the start token) is passed to the decoder. The decoder returns the predictions and the decoder hidden state. The decoder hidden state is then passed back into the model and the predictions are used to calculate the loss. Use teacher forcing to decide the next input to the decoder. Teacher forcing is the technique where the target word is passed as the next input to the decoder. The final step is to calculate the gradients and apply it to the optimizer and backpropagate.

[ ]:

```
import time

loss_plot = []

for epoch in range(0, EPOCHS):
    start = time.time()
    total_loss = 0
```

(continues on next page)

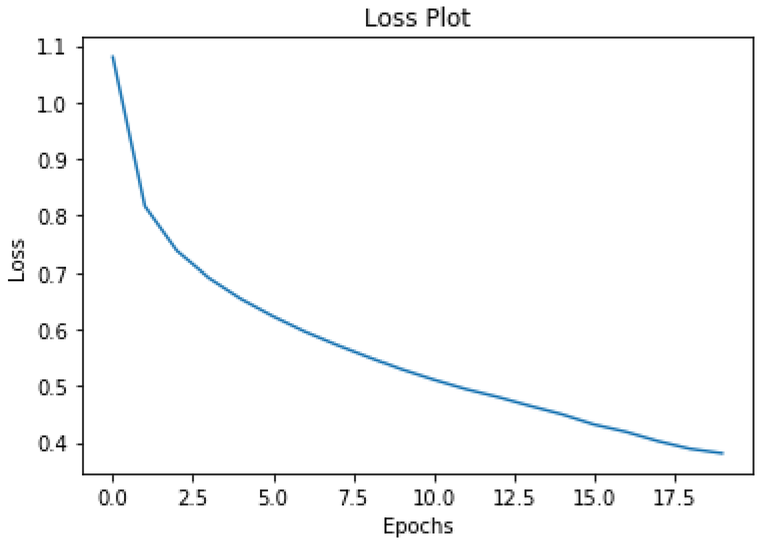
(continued from previous page)

```
    for (batch, (img_tensor, target)) in enumerate(dataset):
        batch_loss, t_loss = train_step(img_tensor, target)
        total_loss += t_loss
        if batch % 1 == 0:
            print('Epoch {} Batch {} Loss {:.4f}'.format(
                epoch + 1, batch, batch_loss.numpy() / int(target.shape[1])))
    # storing the epoch and loss value to plot later
    loss_plot.append(total_loss / num_steps)

    print('Epoch {} Loss {:.6f}'.format(epoch + 1,
                                        total_loss / num_steps))
    print('Time taken for 1 epoch {} sec\n'.format(time.time() - start))
```

Visualize the loss

[23]:
```
import matplotlib.pyplot as plt
# Below loss curve is not the actual loss image we have got
# while training and kept it here only as a reference
plt.plot(loss_plot)
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Loss Plot')
plt.show()
```

[23]:



4.6.6 "Real World" Quick Start Tutorial

This tutorial will guide you on working with the basics of Hangar, while playing with some "real world" data:

- adding data to a repository
- commiting changes
- reading data from a commit
- inspecting contents of a commit

Setup

You can install Hangar via `pip`:

```
$ pip install hangar
```

or via `conda`:

```
$ conda install -c conda-forge hangar
```

Other requirements for this tutorial are:

- pillow - the python imaging library
- tqdm - a simple tool to display progress bars (this is installed automatically as it is a requirement for `Hangar`)

```
$ pip install pillow
```

1. Create and Initialize a "Repository"

When working with Hangar programatically (the CLI is covered in later tutorials), we always start with the following import:

[1]:

```
from hangar import Repository
```

Create the folder where you want to store the Hangar `Repository`:

[2]:

```
!mkdir /Volumes/Archivio/tensorwerk/hangar/imagenette
```

and create the `Repository` object. Note that when you specify a new folder for a Hangar repository, Python shows you a warning saying that you will need to initialize the repo before starting working on it.

[3]:

```
repo = Repository(path="/Volumes/Archivio/tensorwerk/hangar/imagenette")
```

```
//anaconda/envs/hangar-nested/lib/python3.7/site-packages/hangar-0.5.0.dev1-py3.7-
↪macosx-10.9-x86_64.egg/hangar/context.py:94: UserWarning: No repository exists at /
↪Volumes/Archivio/tensorwerk/hangar/imagenette/.hangar, please use `repo.init()`␣
↪method
  warnings.warn(msg, UserWarning)
```

Initialize the `Repository` providing your name and your email.

> **Warning:** Please be aware that the `remove_old` parameter set to `True` **removes and reinitializes** a Hangar repository at the given path.

[4]:

```
repo.init(
    user_name="Alessia Marcolini", user_email="alessia@tensorwerk.com", remove_
↪old=True
)
```

```
Hangar Repo initialized at: /Volumes/Archivio/tensorwerk/hangar/imagenette/.hangar
```

[4]:

```
'/Volumes/Archivio/tensorwerk/hangar/imagenette/.hangar'
```

2. Open the Staging Area for Writing

A `Repository` can be checked out in two modes: write-enabled and read-only. We need to checkout the repo in write mode in order to initialize the columns and write into them.

[5]:

```
master_checkout = repo.checkout(write=True)
```

A checkout allows access to `columns`. The `columns` attribute of a checkout provides the interface to working with all of the data on disk!

[6]:

```
master_checkout.columns
```

[6]:

```
Hangar Columns
    Writeable         : True
    Number of Columns : 0
    Column Names / Partial Remote References:
      -
```

3. Download and Prepare Some Conventionally Stored Data

To start playing with Hangar, let's get some data to work on. We'll be using the Imagenette dataset.

The following commands will download ~96 MB of data to the local directory and decompress the tarball containing ~ 9,200 `.jpeg` images in the folder `data` in the current working directory.

[7]:

```
!wget https://s3.amazonaws.com/fast-ai-imageclas/imagenette2-160.tgz -P data
```

```
--2020-04-04 13:25:37--  https://s3.amazonaws.com/fast-ai-imageclas/imagenette2-160.
↪tgz
Resolving s3.amazonaws.com... 52.216.238.197
Connecting to s3.amazonaws.com|52.216.238.197|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 98948031 (94M) [application/x-tar]
Saving to: ‘data/imagenette2-160.tgz’

imagenette2-160.tgz 100%[===================>]  94.36M  4.52MB/s    in 22s

2020-04-04 13:26:00 (4.31 MB/s) - ‘data/imagenette2-160.tgz’ saved [98948031/98948031]
```

[8]:

```
!tar -xzf data/imagenette2-160.tgz -C data
```

[9]:

```
!wget http://image-net.org/archive/words.txt -P data/imagenette2-160
```

```
--2020-04-04 13:26:24--  http://image-net.org/archive/words.txt
Resolving image-net.org... 171.64.68.16
Connecting to image-net.org|171.64.68.16|:80... connected.
```

(continues on next page)

(continued from previous page)

```
HTTP request sent, awaiting response... 200 OK
Length: 2655750 (2.5M) [text/plain]
Saving to: ‘data/imagenette2-160/words.txt’

words.txt           100%[===================>]   2.53M   884KB/s    in 2.9s

2020-04-04 13:26:27 (884 KB/s) - ‘data/imagenette2-160/words.txt’ saved [2655750/
↪2655750]
```

The dataset directory structure on disk is as follows:

Each subdirectory in the `train` / `val` folders (named starting with `"n0"`) contains a few hundred images which feature objects/elements of a common classification (tench, English springer, cassette player, chain saw, church, French horn, garbage truck, gas pump, golf ball, parachute, etc.). The image file names follow a convention specific to the ImageNet project, but can be thought of as essentially random (so long as they are unique).

```
imagenette2-160
├── train
│   ├── n01440764
│   ├── n02102040
│   ├── n02979186
│   ├── n03000684
│   ├── n03028079
│   ├── n03394916
│   ├── n03417042
│   ├── n03425413
│   ├── n03445777
│   └── n03888257
└── val
    ├── n01440764
    ├── n02102040
    ├── n02979186
    ├── n03000684
    ├── n03028079
    ├── n03394916
    ├── n03417042
    ├── n03425413
    ├── n03445777
    └── n03888257
```

Classification/Label Data

The labels associated with each image are contained in a seperate `.txt` file, we download the `words.txt` to the directory the images are extracted into.

Reviewing the contents of this file, we will find a mapping of classification codes (subdirectory names starting with `"n0"`) to human readable descriptions of the contents. A small selection of the file is provided below as an illustration.

```
n01635343   Rhyacotriton, genus Rhyacotriton
n01635480   olympic salamander, Rhyacotriton olympicus
n01635659   Plethodontidae, family Plethodontidae
n01635964   Plethodon, genus Plethodon
```

(continues on next page)

(continued from previous page)

```
n01636127   lungless salamander, plethodont
n01636352   eastern red-backed salamander, Plethodon cinereus
n01636510   western red-backed salamander, Plethodon vehiculum
n01636675   Desmograthus, genus Desmograthus
n01636829   dusky salamander
n01636984   Aneides, genus Aneides
n01637112   climbing salamander
n01637338   arboreal salamander, Aneides lugubris
n01637478   Batrachoseps, genus Batrachoseps
n01637615   slender salamander, worm salamander
n01637796   Hydromantes, genus Hydromantes
```

Mapping Classification Codes to Meaningful Descriptors

We begin by reading each line of this file and creating a dictionary to store the corrispondence between ImageNet synset name and a human readable label.

[10]:

```python
from pathlib import Path

dataset_dir = Path("./data/imagenette2-160")

synset_label = {}
with open(dataset_dir / "words.txt", "r") as f:
    for line in f.readlines():
        synset, label = line.split("\t")
        synset_label[synset] = label.rstrip()
```

Read training data (images and labels) from disk and store them in NumPy arrays.

[11]:

```python
import os
from tqdm import tqdm

import numpy as np
from PIL import Image
```

[12]:

```python
train_images = []
train_labels = []

for synset in tqdm(os.listdir(dataset_dir / "train")):
    label = synset_label[synset]

    for image_filename in os.listdir(dataset_dir / "train" / synset):
        image = Image.open(dataset_dir / "train" / synset / image_filename)
        image = image.resize((163, 160))
        data = np.asarray(image)

        if len(data.shape) == 2:  # discard B&W images
            continue

        train_images.append(data)
        train_labels.append(label)

train_images = np.array(train_images)
```

```
100%|| 10/10 [00:31<00:00,  3.12s/it]
```

[13]:
```
train_images.shape
```

[13]:
```
(9296, 160, 163, 3)
```

Note: Here we are reading the images from disk and storing them in a big Python list, and then converting it to a NumPy array. Note that it could be impractical for larger datasets. You might want to consider the idea of reading files in batch.

Read validation data (images and labels) from disk and store them in NumPy arrays, same as before.

[14]:
```
val_images = []
val_labels = []

for synset in tqdm(os.listdir(dataset_dir / "val")):
    label = synset_label[synset]

    for image_filename in os.listdir(dataset_dir / "val" / synset):
        image = Image.open(dataset_dir / "val" / synset / image_filename)
        image = image.resize((163, 160))
        data = np.asarray(image)

        if len(data.shape) == 2:  # discard B&W images
            continue

        val_images.append(data)
        val_labels.append(label)

val_images = np.array(val_images)
```

```
100%|| 10/10 [00:12<00:00,  1.22s/it]
```

[15]:
```
val_images.shape
```

[15]:
```
(3856, 160, 163, 3)
```

4. Column initialization

With checkout write-enabled, we can now initialize a new column of the repository using the method `add_ndarray_column()`.

All samples within a column have the same data type, and number of dimensions. The size of each dimension can be either fixed (the default behavior) or variable per sample.

You will need to provide a column `name` and a `prototype`, so Hangar can infer the shape of the elements contained in the array. `train_im_col` will become a column accessor object.

[16]:
```
train_im_col = master_checkout.add_ndarray_column(
    name="training_images", prototype=train_images[0]
)
```

Verify we successfully added the new column:

[17]:
```
master_checkout.columns
```

[17]:
```
Hangar Columns
    Writeable         : True
    Number of Columns : 1
    Column Names / Partial Remote References:
      - training_images / False
```

Get useful information about the new column simply by inspecting `train_im_col` ...

[18]:
```
train_im_col
```

[18]:
```
Hangar FlatSampleWriter
    Column Name              : training_images
    Writeable                : True
    Column Type              : ndarray
    Column Layout            : flat
    Schema Type              : fixed_shape
    DType                    : uint8
    Shape                    : (160, 163, 3)
    Number of Samples        : 0
    Partial Remote Data Refs : False
```

... or by leveraging the dict-style columns access through the `checkout` object. They provide the same information.

[19]:
```
master_checkout.columns["training_images"]
```

[19]:
```
Hangar FlatSampleWriter
    Column Name              : training_images
    Writeable                : True
    Column Type              : ndarray
    Column Layout            : flat
    Schema Type              : fixed_shape
    DType                    : uint8
    Shape                    : (160, 163, 3)
    Number of Samples        : 0
    Partial Remote Data Refs : False
```

Since Hangar 0.5, it’s possible to have a column with string datatype, and we will be using it to store the labels of our dataset.

[20]:
```
train_lab_col = master_checkout.add_str_column(name="training_labels")
```

[21]:
```
train_lab_col
```

[21]:
```
Hangar FlatSampleWriter
    Column Name              : training_labels
    Writeable                : True
    Column Type              : str
    Column Layout            : flat
    Schema Type              : variable_shape
    DType                    : <class 'str'>
    Shape                    : None
    Number of Samples        : 0
    Partial Remote Data Refs : False
```

5. Adding data

To add data to a named column, we can use dict-style mode (refer to the __setitem__, __getitem__, and __delitem__ methods) or the update() method. Sample keys can be either str or int type.

[22]:
```
train_im_col[0] = train_images[0]
train_lab_col[0] = train_labels[0]
```

As we can see, Number of Samples is equal to 1 now.

[23]:
```
master_checkout.columns["training_labels"]
```

[23]:
```
Hangar FlatSampleWriter
    Column Name              : training_labels
    Writeable                : True
    Column Type              : str
    Column Layout            : flat
    Schema Type              : variable_shape
    DType                    : <class 'str'>
    Shape                    : None
    Number of Samples        : 1
    Partial Remote Data Refs : False
```

[24]:
```
data = {1: train_images[1], 2: train_images[2]}
```

[25]:
```
train_im_col.update(data)
```

[26]:
```
train_im_col
```

[26]:
```
Hangar FlatSampleWriter
    Column Name              : training_images
    Writeable                : True
    Column Type              : ndarray
    Column Layout            : flat
    Schema Type              : fixed_shape
    DType                    : uint8
    Shape                    : (160, 163, 3)
    Number of Samples        : 3
    Partial Remote Data Refs : False
```

Let's add the remaining training images:

[27]:
```
with train_im_col:
    for i, img in tqdm(enumerate(train_images), total=train_images.shape[0]):
        if i not in [0, 1, 2]:
            train_im_col[i] = img
```

```
100%|| 9296/9296 [00:36<00:00, 257.92it/s]
```

[28]:
```
with train_lab_col:
    for i, label in tqdm(enumerate(train_labels), total=len(train_labels)):
        if i != 0:
            train_lab_col[i] = label
```

```
100%|| 9296/9296 [00:01<00:00, 5513.23it/s]
```

[29]:
```
train_lab_col
```

[29]:
```
Hangar FlatSampleWriter
    Column Name              : training_labels
    Writeable                : True
    Column Type              : str
    Column Layout            : flat
    Schema Type              : variable_shape
    DType                    : <class 'str'>
    Shape                    : None
    Number of Samples        : 9296
    Partial Remote Data Refs : False
```

Both the `training_images` and the `training_labels` have 9296 samples. Great!

Note: To get an overview of the different ways you could add data to a Hangar repository (also from a performance point of view), please refer to the Performance section of the Hangar Tutorial Part 1.

6. Committing changes

Once you have made a set of changes you want to commit, simply call the `commit()` method and specify a message.

The returned value (`a=ecc943c89b9b09e41574c9849f11937828fece28`) is the commit hash of this commit.

[30]:
```
master_checkout.commit("Add Imagenette training images and labels")
```

[30]:
```
'a=ecc943c89b9b09e41574c9849f11937828fece28'
```

Let's add the validation data to the repository . . .

[31]:
```
val_im_col = master_checkout.add_ndarray_column(
    name="validation_images", prototype=val_images[0]
)
val_lab_col = master_checkout.add_str_column(name="validation_labels")
```

[32]:
```
with val_im_col, val_lab_col:
    for img, label in tqdm(zip(val_images, val_labels), total=len(val_labels)):
        val_im_col[i] = img
        val_lab_col[i] = label
```

```
100%|| 3856/3856 [00:08<00:00, 474.25it/s]
```

. . . and commit!

[33]:
```
master_checkout.commit("Add Imagenette validation images and labels")
```

[33]:
```
'a=e31ef9a06c8d1a4cefeb52c336b2c33d1dca3fba'
```

To view the **history** of your commits:

[34]:
```
master_checkout.log()
```

```
* a=e31ef9a06c8d1a4cefeb52c336b2c33d1dca3fba (master) : Add Imagenette validation␣
↪images and labels
* a=ecc943c89b9b09e41574c9849f11937828fece28 : Add Imagenette training images and␣
↪labels
```

(continues on next page)

(continued from previous page)

Do not forget to close the write-enabled checkout!

[35]:
```
master_checkout.close()
```

Let's inspect the repository state! This will show disk usage information, the details of the last commit and all the information about the dataset columns.

[36]:
```
repo.summary()
```

```
Summary of Contents Contained in Data Repository

==================
| Repository Info
|-----------------
|  Base Directory: /Volumes/Archivio/tensorwerk/hangar/imagenette
|  Disk Usage: 862.09 MB

===================
| Commit Details
-------------------
|  Commit: a=e31ef9a06c8d1a4cefeb52c336b2c33d1dca3fba
|  Created: Sat Apr  4 11:29:12 2020
|  By: Alessia Marcolini
|  Email: alessia@tensorwerk.com
|  Message: Add Imagenette validation images and labels

==================
| DataSets
|-----------------
|  Number of Named Columns: 4
|
|  * Column Name: ColumnSchemaKey(column="training_images", layout="flat")
|    Num Data Pieces: 9296
|    Details:
|    - column_layout: flat
|    - column_type: ndarray
|    - schema_type: fixed_shape
|    - shape: (160, 163, 3)
|    - dtype: uint8
|    - backend: 01
|    - backend_options: {'complib': 'blosc:lz4hc', 'complevel': 5, 'shuffle': 'byte'}
|
|  * Column Name: ColumnSchemaKey(column="training_labels", layout="flat")
|    Num Data Pieces: 9296
|    Details:
|    - column_layout: flat
|    - column_type: str
|    - schema_type: variable_shape
|    - dtype: <class'str'>
|    - backend: 30
|    - backend_options: {}
|
|  * Column Name: ColumnSchemaKey(column="validation_images", layout="flat")
```

(continues on next page)

(continued from previous page)

```
|    Num Data Pieces: 1
|    Details:
|    - column_layout: flat
|    - column_type: ndarray
|    - schema_type: fixed_shape
|    - shape: (160, 163, 3)
|    - dtype: uint8
|    - backend: 01
|    - backend_options: {'complib': 'blosc:lz4hc', 'complevel': 5, 'shuffle': 'byte'}
|
|  * Column Name: ColumnSchemaKey(column="validation_labels", layout="flat")
|    Num Data Pieces: 1
|    Details:
|    - column_layout: flat
|    - column_type: str
|    - schema_type: variable_shape
|    - dtype: <class'str'>
|    - backend: 30
|    - backend_options: {}

==================
| Metadata:
|-----------------
|  Number of Keys: 0
```

Great! You've made it until the end of the "Real World" Quick Start Tutorial!!

Please check out the other tutorials for more advanced stuff such as branching & merging, conflicts resolution and data loaders for TensorFlow and PyTorch!

4.7 Hangar Under The Hood

At its core, Hangar is a content addressable data store whose design requirements were inspired by the Git version control system.

4.7.1 Things In Life Change, Your Data Shouldn't

When designing a high performance data version control system, achieving performance goals while ensuring consistency is incredibly difficult. Memory is fast, disk is slow; not much we can do about it. But since Hangar should deal with any numeric data in an array of any size (with an enforced limit of 31 dimensions in a sample…) we have to find ways to work *with* the disk, not against it.

Upon coming to terms with this face, we are actually presented with a problem once we realize that we live in the real world, and real world is ugly. Computers crash, processes get killed, and people do * *interesting* * things. Because of this, It is a foundational design principle for us to **guarantee that once Hangar says data has been successfully added to the repository, it is actually persisted.** This essentially means that any process which interacts with data records on disk must be stateless. If (for example) we were to keep a record of all data added to the staging area in an in-memory list, and the process gets killed, we may have just lost references to all of the array data, and may not even be sure that the arrays were flushed to disk properly. These situations are a NO-GO from the start, and will always remain so.

So, we come to the first design choice: **read and write actions are atomic**. Once data is added to a Hangar repository, the numeric array along with the necessary book-keeping records will *always* occur transactionally, ensuring that when something unexpected happens, the data and records are committed to disk.

Note: The atomicity of interactions is completely hidden from a normal user; they shouldn't have to care about this or even know this exists. However, this is also why using the context-manager style column interaction scheme can result in ~2x times speedup on writes/reads. We can just pass on most of the work to the Python `contextlib` package instead of having to begin and commit/abort (depending on interaction mode) transactions with every call to an *add* or *get* method.

4.7.2 Data Is Large, We Don't Waste Space

From the very beginning we knew that while it would be easy to just store all data in every commit as independent arrays on disk, such a naive implementation would just absolutely eat up disk space for any repository with a non-trivial history. Hangar commits should be fast and use minimal disk space, duplicating data just doesn't make sense for such a system. And so we decided on implementing a content addressable data store backend.

When a user requests to add data to a Hangar repository, one of the first operations which occur is to generate a hash of the array contents. If the hash does not match a piece of data already placed in the Hangar repository, the data is sent to the appropriate storage backend methods. On success, the backend sends back some arbitrary specification which can be used to retrieve that same piece of data from that particular backend. The record backend then stores a key/value pair of (*hash*, *backend_specification*).

Note: The record backend stores hash information in a separate location from the commit references (which associate a *(columnname, sample name/id)* to a *sample_hash*). This let's us separate the historical repository information from a particular computer's location of a data piece. All we need in the public history is to know that some data with a particular hash is associated with a commit. No one but the system which actually needs to access the data needs to know where it can be found.

On the other hand, if a data sample is added to a repository which already has a record of some hash, we don't even involve the storage backend. All we need to do is just record that a new sample in a column was added with that hash. It makes no sense to write the same data twice.

This method can actually result in massive space savings for some common use cases. For the MNIST column, the training label data is typically a 1D-array of size 50,000. Because there are only 10 labels, we only need to store 10 ints on disk, and just keep references to the rest.

4.7.3 The Basics of Collaboration: Branching and Merging

Up to this point, we haven't actually discussed much about how data and records are treated on disk. We'll leave an entire walkthrough of the backend record structure for another tutorial, but let's introduce the basics here, and see how we enable the types of branching and merging operations you might be used to with source code (at largely the same speed!).

Here's a few core principles to keep in mind:

Numbers == Numbers

Hangar has no concept of what a piece of data is outside of a string of bytes / numerical array, and most importantly, *hangar does not care*; Hangar is a tool, and we leave it up to you to know what your data actually means)!

At the end of the day when the data is placed into *some* collection on disk, the storage backend we use won't care either. In fact, this is the entire reason why Hangar can do what it can; we don't attempt to treat data as anything other then a series of bytes on disk!

The fact that *Hangar does not care about what your data represents* is a fundamental underpinning of how the system works under the hood. It is the *designed and intended behavior* of Hangar to dump arrays to disk in what would seem like completely arbitrary buffers/locations to an outside observer. And for the most part, they would be essentially correct in their observation that data samples on disk are in strange locations.

While there is almost no organization or hierarchy for the actual data samples when they are stored on disk, that is not to say that they are stored without care! We may not care about global trends, but we do care a great deal about the byte order/layout, sequentiality, chunking/compression and validations operations which are applied across the bytes which make up a data sample.

In other words, we optimize for utility and performance on the backend, not so that a human can understand the file format without a computer! After the array has been saved to disk, all we care about is that bookkeeper can record some unique information about where some piece of content is, and how we can read it. **None of that information is stored alongside the data itself - Remember: numbers are just numbers - they don't have any concept of what they are**.

Records != Numbers

The form numerical data takes once dumped on disk is completely irrelevant to the specifications of records in the repository history.

Now, let's unpack this for a bit. We know from *Numbers == Numbers* that data is saved to disk in some arbitrary locations with some arbitrary backend. We also know from *Data Is Large, We Don't Waste Space* that the permanent repository information only contains a record which links a sample name to a hash. We also assert that there is also a mapping of hash to storage backend specification kept somewhere (doesn't matter what that mapping is for the moment). With those 3 pieces of information, it's obvious that once data is placed in the repository, we don't actually need to interact with it to understand the accounting of what was added when!

In order to make a commit, we just pack up all the records which existed in the staging area, create a hash of the records (including the hash of any parent commits), and then store the commit hash mapping alongside details such as the commit user/email and commit message, and a compressed version of the full commit records as they existed at that point in time.

Note: That last point "storing a compressed version of the full commit records", is semi inefficient, and will be changed in the future so that unchanged records are note duplicated across commits.

An example is given below of the keys -> values mapping which stores each of the staged records, and which are packed up / compressed on commit (and subsequently unpacked on checkout!).

```
Num asets                      'a.'               -> '2'
---------------------------------------------------------------------------
Name of aset -> num samples || 'a.train_images'   -> '10'
Name of data -> hash        || 'a.train_images.0' -> BAR_HASH_1'
Name of data -> hash        || 'a.train_images.1' -> BAR_HASH_2'
Name of data -> hash        || 'a.train_images.2' -> BAR_HASH_3'
Name of data -> hash        || 'a.train_images.3' -> BAR_HASH_4'
```

(continues on next page)

(continued from previous page)

```
Name of data -> hash        || 'a.train_images.4' -> BAR_HASH_5'
Name of data -> hash        || 'a.train_images.5' -> BAR_HASH_6'
Name of data -> hash        || 'a.train_images.6' -> BAR_HASH_7'
Name of data -> hash        || 'a.train_images.7' -> BAR_HASH_8'
Name of data -> hash        || 'a.train_images.8' -> BAR_HASH_9'
Name of data -> hash        || 'a.train_images.9' -> BAR_HASH_0'
---------------------------------------------------------------------------
Name of aset -> num samples || 'a.train_labels'   -> '10'
Name of data -> hash        || 'a.train_labels.0' -> BAR_HASH_11'
Name of data -> hash        || 'a.train_labels.1' -> BAR_HASH_12'
Name of data -> hash        || 'a.train_labels.2' -> BAR_HASH_13'
Name of data -> hash        || 'a.train_labels.3' -> BAR_HASH_14'
Name of data -> hash        || 'a.train_labels.4' -> BAR_HASH_15'
Name of data -> hash        || 'a.train_labels.5' -> BAR_HASH_16'
Name of data -> hash        || 'a.train_labels.6' -> BAR_HASH_17'
Name of data -> hash        || 'a.train_labels.7' -> BAR_HASH_18'
Name of data -> hash        || 'a.train_labels.8' -> BAR_HASH_19'
Name of data -> hash        || 'a.train_labels.9' -> BAR_HASH_10'
---------------------------------------------------------------------------
's.train_images'   -> '{"schema_hash": "RM4DefFsjRs=",
                        "schema_dtype": 2,
                        "schema_is_var": false,
                        "schema_max_shape": [784],
                        "schema_is_named": true}'
's.train_labels'   -> '{"schema_hash":
                        "ncbHqE6Xldg=",
                        "schema_dtype": 7,
                        "schema_is_var": false,
                        "schema_max_shape": [1],
                        "schema_is_named": true}'
```

History is Relative

Though it may be a bit obvious to state, it is of critical importance to realize that it is only because we store the full contents of the repository staging area as it existed in the instant just prior to a commit, that the integrity of full repository history can be verified from a single commit's contents and expected hash value. More so, any single commit has only a topical relationship to a commit at any other point in time. It is only our imposition of a commit's ancestry tree which actualizes any subsequent insights or interactivity

While the general process of topological ordering: create branch, checkout branch, commit a few times, and merge, follows the *git* model fairly well at a conceptual level, there are some important differences we want to highlight due to their implementation differences:

1) Multiple commits can simultaneously checked out in "read-only" mode on a single machine. Checking out a commit for reading does not touch the staging area status.
2) Only one process can interact with the a write-enabled checkout at a time.
3) A detached head CANNOT exist for write enabled checkouts. A staging area must begin with an identical state to the most recent commit of a/any branch.
4) A staging area which has had changes made in it cannot switch base branch without either a commit, hard-reset, or (soon to be developed) stash operation.

When a repository is initialized, a record is created which indicates the staging area's *HEAD* branch. in addition, a branch is created with the name *master*, and which is the only commit in the entire repository which will have no parent. The record key/value pairs resemble the following:

```
'branch.master' -> ''                # No parent commit.
'head'          -> 'branch.master'   # Staging area head branch

# Commit Hash  |  Parent Commit
-------------------------------------
```

> **Warning:** Much like git, odd things can happen before the *'initial commit'* is made. We recommend creating the initial commit as quickly as possible to prevent undefined behavior during repository setup. In the future, we may decide to create the "initial commit" automatically upon repository initialization.

Once the initial commit is made, a permanent commit record in made which specifies the records (not shown below) and the parent commit. The branch head pointer is then updated to point to that commit as it's base.

```
'branch.master' -> '479b4cfff6219e3d'
'head'          -> 'branch.master'

# Commit Hash       |  Parent Commit
-------------------------------------
'479b4cfff6219e3d' ->  ''
```

Branches can be created as cheaply as a single line of text can be written, and they simply require a "root" commit hash (or a branch name, in which case the branch's current HEAD commit will be used as the root HEAD). Likewise a branch can be merged with just a single write operation (once the merge logic has completed - a process which is explained separately from this section; just trust that it happens for now).

A more complex example which creates 4 different branches and merges them in a complicated order can be seen below. Please note that the `` << `` symbol is used to indicate a merge commit where *X* << *Y* reads: `'merging dev branch Y into master branch X'`.

```
'branch.large_branch' -> '8eabd22a51c5818c'
'branch.master'       -> '2cd30b98d34f28f0'
'branch.test_branch'  -> '1241a36e89201f88'
'branch.trydelete'    -> '51bec9f355627596'
'head'                -> 'branch.master'

 # Commit Hash       |  Parent Commit
 -------------------------------------
'1241a36e89201f88'  -> '8a6004f205fd7169'
'2cd30b98d34f28f0'  -> '9ec29571d67fa95f << 51bec9f355627596'
'51bec9f355627596'  -> 'd683cbeded0c8a89'
'69a09d87ea946f43'  -> 'd683cbeded0c8a89'
'8a6004f205fd7169'  -> 'a320ae935fc3b91b'
'8eabd22a51c5818c'  -> 'c1d596ed78f95f8f'
'9ec29571d67fa95f'  -> '69a09d87ea946f43 << 8eabd22a51c5818c'
'a320ae935fc3b91b'  -> 'e3e79dd897c3b120'
'c1d596ed78f95f8f'  -> ''
'd683cbeded0c8a89'  -> 'fe0bcc6a427d5950 << 1241a36e89201f88'
'e3e79dd897c3b120'  -> 'c1d596ed78f95f8f'
'fe0bcc6a427d5950'  -> 'e3e79dd897c3b120'
```

Because the raw commit hash logs can be quite dense to parse, a graphical logging utility is included as part of the repository. Running the `Repository.log()` method will pretty print a graph representation of the commit history:

```
>>> from hangar import Repository
>>> repo = Repository(path='/foo/bar/path/')
```

(continues on next page)

(continued from previous page)

```
... # make some commits

>>> repo.log()
```

```
*   2cd30b98d34f28f0 (31Mar2019 16:26:31) (test user): try number two
|\
* \   9ec29571d67fa95f (31Mar2019 16:26:31) (test user): merging the long running branch into master
|\ \
| | * 51bec9f355627596 (31Mar2019 16:26:31) (test user): another on try delete
* | | 69a09d87ea946f43 (31Mar2019 16:26:31) (test user): another on master
| |/
|/|
* |   d683cbeded0c8a89 (31Mar2019 16:26:31) (test user): this is the first merge
|\ \
| * | 1241a36e89201f88 (31Mar2019 16:26:31) (test user): another commit on test banch after adding to new_set
| * | 8a6004f205fd7169 (31Mar2019 16:26:31) (test user): second commit on test branch with new dset
| * | a320ae935fc3b91b (31Mar2019 16:26:30) (test user): first commit on test branch
* | | fe0bcc6a427d5950 (31Mar2019 16:26:30) (test user): third commit on master
|/ /
* | e3e79dd897c3b120 (31Mar2019 16:26:30) (test user): second commit on master with training labels
| * 8eabd22a51c5818c (31Mar2019 16:26:22) (test user): first commit on the large branch
|/
* c1d596ed78f95f8f (31Mar2019 16:26:22) (test user): initial commit on master with training images
```

4.8 Hangar CLI Documentation

The CLI described below is automatically available after the Hangar Python package has been installed (either through a package manager or via source builds). In general, the commands require the terminals `cwd` to be at the same level the repository was initially created in.

Simply start by typing `$ hangar --help` in your terminal to get started!

4.8.1 hangar

```
hangar [OPTIONS] COMMAND [ARGS]...
```

Options

--version
display current Hangar Version

branch

Operate on and list branch pointers.

```
hangar branch [OPTIONS] COMMAND [ARGS]...
```

create

Create a branch with NAME at STARTPOINT (short-digest or branch)

If no STARTPOINT is provided, the new branch is positioned at the HEAD of the staging area branch, automatically.

```
hangar branch create [OPTIONS] NAME [STARTPOINT]
```

Arguments

`NAME`
Required argument

`STARTPOINT`
Optional argument

delete

Remove a branch pointer with the provided NAME.

The NAME must be a branch present on the local machine.

```
hangar branch delete [OPTIONS] NAME
```

Options

`-f, --force`
flag to force delete branch which has un-merged history.

Arguments

`NAME`
Required argument

list

List all branch names.

Includes both remote branches as well as local branches.

```
hangar branch list [OPTIONS]
```

checkout

Checkout writer head branch at BRANCHNAME.

This method requires that no process currently holds the writer lock. In addition, it requires that the contents of the staging area are ‘CLEAN’ (no changes have been staged).

```
hangar checkout [OPTIONS] BRANCHNAME
```

Arguments

BRANCHNAME
Required argument

clone

Initialize a repository at the current path and fetch updated records from REMOTE.

Note: This method does not actually download the data to disk. Please look into the `fetch-data` command.

```
hangar clone [OPTIONS] REMOTE
```

Options

--name `<name>`
first and last name of user

--email `<email>`
email address of the user

--overwrite
overwrite a repository if it exists at the current path

Arguments

REMOTE
Required argument

column

Operations for working with columns in the writer checkout.

```
hangar column [OPTIONS] COMMAND [ARGS]...
```

create

Create an column with NAME and DTYPE of SHAPE.

The column will be created in the staging area / branch last used by a writer-checkout. Valid NAMEs contain only ascii letters and [`'.'`, `'_'`, `'-'`] (no whitespace). The DTYPE must be one of [`'UINT8'`, `'INT8'`, `'UINT16'`, `'INT16'`, `'UINT32'`, `'INT32'`, `'UINT64'`, `'INT64'`, `'FLOAT16'`, `'FLOAT32'`, `'FLOAT64'`, `'STR'`].

If a ndarray dtype is specified (not 'STR'), then the SHAPE must be the last argument(s) specified, where each dimension size is identified by a (space seperated) list of numbers.

Examples:

To specify, a column for some training images of dtype uint8 and shape (256, 256, 3) we should say:

```
$ hangar column create train_images UINT8 256 256 3
```

To specify that the samples can be variably shaped (have any dimension size up to the maximum SHAPE specified) we would say:

```
$ hangar column create train_images UINT8 256 256 3 --variable-shape
```

or equivalently:

```
$ hangar column create --variable-shape train_images UINT8 256 256 3
```

To specify that the column contains a nested set of subsample data under a common sample key, the `--contains-subsamples` flag can be used.

```
$ hangar column create --contains-subsamples train_images UINT8 256 256 3
```

```
hangar column create [OPTIONS] NAME [UINT8|INT8|UINT16|INT16|UINT32|INT32|UINT
                     64|INT64|FLOAT16|FLOAT32|FLOAT64|STR] [SHAPE]...
```

Options

`--variable-shape`
flag indicating sample dimensions can be any size up to max shape.

`--contains-subsamples`
flag indicating if this is a column which nests multiple subsamples under a common sample key.

Arguments

`NAME`
Required argument

`DTYPE`
Required argument

`SHAPE`
Optional argument(s)

remove

Delete the column NAME (and all samples) from staging area.

The column will be removed from the staging area / branch last used by a writer-checkout.

```
hangar column remove [OPTIONS] NAME
```

Arguments

NAME
Required argument

commit

Commits outstanding changes.

Commit changes to the given files into the repository. You will need to 'push' to push up your changes to other repositories.

```
hangar commit [OPTIONS]
```

Options

-m, --message <message>
The commit message. If provided multiple times each argument gets converted into a new line.

diff

Display diff of DEV commit/branch to MASTER commit/branch.

If no MASTER is specified, then the staging area branch HEAD will will be used as the commit digest for MASTER. This operation will return a diff which could be interpreted as if you were merging the changes in DEV into MASTER.

TODO: VERIFY ORDER OF OUTPUT IS CORRECT.

```
hangar diff [OPTIONS] DEV [MASTER]
```

Arguments

DEV
Required argument

MASTER
Optional argument

export

Export COLUMN sample data as it existed a STARTPOINT to some format and path.

Specifying which sample to be exported is possible by using the switch --sample (without this, all the samples in the given column will be exported). Since hangar supports both int and str datatype for the sample name, specifying that while mentioning the sample name might be necessary at times. It is possible to do that by separating the name and type by a colon.

Example:

1. if the sample name is string of numeric 10 - str:10 or 10
2. if the sample name is sample1 - str:sample1 or sample1
3. if the sample name is an int, let say 10 - int:10

```
hangar export [OPTIONS] COLUMN [STARTPOINT]
```

Options

-o, --out <outdir>
Directory to export data

-s, --sample <sample>
Sample name to export. Default implementation is to interpret all input names as string type. As a column can contain samples with both str and int types, we allow you to specify name type of the sample. To identify a potentially ambiguous name, we allow you to prepend the type of sample name followed by a colon and then the sample name (ex. str:54 or int:54). This can be done for any sample key.

-f, --format <format_>
File format of output file

--plugin <plugin>
override auto-inferred plugin

Arguments

COLUMN
Required argument

STARTPOINT
Optional argument

fetch

Retrieve the commit history from REMOTE for BRANCH.

This method does not fetch the data associated with the commits. See fetch-data to download the tensor data corresponding to a commit.

```
hangar fetch [OPTIONS] REMOTE BRANCH
```

Arguments

REMOTE
Required argument

BRANCH
Required argument

fetch-data

Get data from REMOTE referenced by STARTPOINT (short-commit or branch).

The default behavior is to only download a single commit's data or the HEAD commit of a branch. Please review optional arguments for other behaviors.

```
hangar fetch-data [OPTIONS] REMOTE STARTPOINT
```

Options

-d, --column <column>
specify any number of column keys to fetch data for.

-n, --nbytes <nbytes>
total amount of data to retrieve in MB/GB.

-a, --all-history
Retrieve data referenced in every parent commit accessible to the STARTPOINT

Arguments

REMOTE
Required argument

STARTPOINT
Required argument

import

Import file or directory of files at PATH to COLUMN in the staging area.

If passing in a directory, all files in the directory will be imported, if passing in a file, just that files specified will be imported.

```
hangar import [OPTIONS] COLUMN PATH
```

Options

--branch <branch>
branch to import data

--plugin <plugin>
override auto-infered plugin

--overwrite
overwrite data samples with the same name as the imported data file

Arguments

COLUMN
Required argument

PATH
Required argument

init

Initialize an empty repository at the current path.

```
hangar init [OPTIONS]
```

Options

--name <name>
first and last name of user

--email <email>
email address of the user

--overwrite
overwrite a repository if it exists at the current path

log

Display commit graph starting at STARTPOINT (short-digest or name)

If no argument is passed in, the staging area branch HEAD will be used as the starting point.

```
hangar log [OPTIONS] [STARTPOINT]
```

Arguments

STARTPOINT
Optional argument

push

Upload local BRANCH commit history / data to REMOTE server.

```
hangar push [OPTIONS] REMOTE BRANCH
```

Arguments

REMOTE
Required argument

BRANCH
Required argument

remote

Operations for working with remote server references

```
hangar remote [OPTIONS] COMMAND [ARGS]...
```

add

Add a new remote server NAME with url ADDRESS to the local client.

This name must be unique. In order to update an old remote, please remove it and re-add the remote NAME / ADDRESS combination.

```
hangar remote add [OPTIONS] NAME ADDRESS
```

Arguments

NAME
Required argument

ADDRESS
Required argument

list

List all remote repository records.

```
hangar remote list [OPTIONS]
```

remove

Remove the remote server NAME from the local client.

This will not remove any tracked remote reference branches.

```
hangar remote remove [OPTIONS] NAME
```

Arguments

NAME

Required argument

server

Start a hangar server, initializing one if does not exist.

The server is configured to top working in 24 Hours from the time it was initially started. To modify this value, please see the `--timeout` parameter.

The hangar server directory layout, contents, and access conventions are similar, though significantly different enough to the regular user "client" implementation that it is not possible to fully access all information via regular API methods. These changes occur as a result of the uniformity of operations promised by both the RPC structure and negotiations between the client/server upon connection.

More simply put, we know more, so we can optimize access more; similar, but not identical.

```
hangar server [OPTIONS]
```

Options

--overwrite

overwrite the hangar server instance if it exists at the current path.

--ip `<ip>`

the ip to start the server on. default is *localhost*

Default localhost

--port `<port>`

port to start the server on. default in *50051*

Default 50051

--timeout `<timeout>`

time (in seconds) before server is stopped automatically

Default 86400

status

Display changes made in the staging area compared to its base commit.

```
hangar status [OPTIONS]
```

summary

Display content summary at STARTPOINT (short-digest or branch).

If no argument is passed in, the staging area branch HEAD wil be used as the starting point. In order to recieve a machine readable, and more complete version of this information, please see the `Repository.summary()` method of the API.

```
hangar summary [OPTIONS] [STARTPOINT]
```

Arguments

`STARTPOINT`
Optional argument

view

Use a plugin to view the data of some SAMPLE in COLUMN at STARTPOINT.

```
hangar view [OPTIONS] COLUMN SAMPLE [STARTPOINT]
```

Options

`-f, --format` `<format_>`
File format of output file

`--plugin` `<plugin>`
Plugin name to use instead of auto-inferred plugin

Arguments

`COLUMN`
Required argument

`SAMPLE`
Required argument

`STARTPOINT`
Optional argument

writer-lock

Determine if the writer lock is held for a repository.

Passing the `--force-release` flag will instantly release the writer lock, invalidating any process which currently holds it.

```
hangar writer-lock [OPTIONS]
```

Options

--force-release
force release writer lock from the CLI.

4.9 Hangar External

High level interaction interface between hangar and everything external.

4.9.1 High Level Methods

High level methods let user interact with hangar without diving into the internal methods of hangar. We have enabled four basic entry points as high level methods

1. `load()`
2. `save()`
3. `show()`
4. `board_show()`

These entry points by itself is not capable of doing anything. But they are entry points to the same methods in the *hangar.external* plugins available in pypi. These high level entry points are used by the CLI for doing import, export and view operations as well as the hangarboard for visualization (using `board_show`)

board_show(*arr: numpy.ndarray*, *plugin: str = None*, *extension: str = None*, ***plugin_kwargs*)
Wrapper to convert the numpy array using the `board_show` method of the plugin to make it displayable in the web UI

Parameters

- **arr** (`numpy.ndarray`) – Data to process into some human understandable representation.
- **plugin** (*str, optional*) – Name of plugin to use. By default, the preferred plugin for the given file format tried until a suitable. This cannot be `None` if `extension` is also `None`
- **extension** (*str, optional*) – Format of the file. This is used to infer which plugin to use in case plugin name is not provided. This cannot be `None` if `plugin` is also `None`

Other Parameters plugin_kwargs (*dict*) – Plugin specific keyword arguments. If the function is being called from command line argument, all the unknown keyword arguments will be collected as `plugin_kwargs`

load(*fpath: str*, *plugin: str = None*, *extension: str = None*, ***plugin_kwargs*) → Tuple[numpy.ndarray, str]
Wrapper to load data from file into memory as numpy arrays using plugin's *load* method

Parameters

- **fpath** (*str*) – Data file path, e.g. `path/to/test.jpg`
- **plugin** (*str, optional*) – Name of plugin to use. By default, the preferred plugin for the given file format tried until a suitable. This cannot be *None* if *extension* is also *None*
- **extension** (*str, optional*) – Format of the file. This is used to infer which plugin to use in case plugin name is not provided. This cannot be *None* if *plugin* is also *None*

Other Parameters plugin_kwargs (*dict*) – Plugin specific keyword arguments. If the function is being called from command line argument, all the unknown keyword arguments will be collected as `plugin_kwargs`

Returns img_array – data returned from the given plugin.

Return type `numpy.ndarray`

save(*arr: numpy.ndarray*, *outdir: str*, *sample_det: str*, *extension: str*, *plugin: str = None*, ***plugin_kwargs*)

Wrapper plugin `save` methods which dump `numpy.ndarray` to disk.

Parameters

- **arr** (`numpy.ndarray`) – Numpy array to be saved to file
- **outdir** (*str*) – Target directory
- **sample_det** (*str*) – Sample name and type of the sample name formatted as `sample_name_type:sample_name`
- **extension** (*str*) – Format of the file. This is used to infer which plugin to use in case plugin name is not provided. This cannot be `None` if `plugin` is also `None`
- **plugin** (*str, optional*) – Name of plugin to use. By default, the preferred plugin for the given file format tried until a suitable. This cannot be `None` if `extension` is also `None`

Other Parameters plugin_kwargs (*dict*) – Plugin specific keyword arguments. If the function is being called from command line argument, all the unknown keyword arguments will be collected as `plugin_kwargs`

Notes

CLI or this method does not create the file name where to save. Instead they pass the required details downstream to the plugins to do that once they verify the given `outdir` is a valid directory. It is because we expect to get data entries where one data entry is one file (like images) and also data entries where multiple entries goes to single file (like CSV). With these ambiguous cases in hand, it's more sensible to let the plugin handle the file handling accordingly.

show(*arr: numpy.ndarray*, *plugin: str = None*, *extension: str = None*, ***plugin_kwargs*)

Wrapper to display `numpy.ndarray` via plugin `show` method.

Parameters

- **arr** (`numpy.ndarray`) – Data to process into some human understandable representation.
- **plugin** (*str, optional*) – Name of plugin to use. By default, the preferred plugin for the given file format tried until a suitable. This cannot be `None` if `extension` is also `None`
- **extension** (*str, optional*) – Format of the file. This is used to infer which plugin to use in case plugin name is not provided. This cannot be `None` if `plugin` is also `None`

Other Parameters **plugin_kwargs** (*dict*) – Plugin specific keyword arguments. If the function is being called from command line argument, all the unknown keyword arguments will be collected as `plugin_kwargs`

4.9.2 Plugin System

Hangar's external plugin system is designed to make it flexible for users to write custom plugins for custom data formats. External plugins should be python installables and should make itself discoverable using package meta data. A detailed documentation can be found in the official python doc. But for a headstart and to avoid going through this somewhat complex process, we have made a cookiecutter package. All the hangar plugins follow the naming standard similar to Flask plugins i.e *hangar_pluginName*

class BasePlugin(*provides*, *accepts*)

Base plugin class from where all the external plugins should be inherited.

Child classes can have four methods to expose - `load`, `save`, `show` and `board_show`. These are considered as valid methods and should be passed as the first argument while initializing the parent from child. Child should also inform the parent about the acceptable file formats by passing that as second argument. *`BasePlugin`* accepts `provides` and `accepts` on init and exposes them which is then used by plugin manager while loading the modules. BasePlugin also provides `sample_name` function to figure out the sample name from the file path. This function is used by `load` method to return the sample name which is then used by hangar as a key to save the data

board_show(*arr*, **args*, ***kwargs*)

Show/display data in hangarboard format.

Hangarboard is capable of displaying three most common data formats: image, text and audio. This function should process the input `numpy.ndarray` data and convert it to any of the supported formats.

load(*fpath*, **args*, ***kwargs*)

Load some data file on disk to recover it in `numpy.ndarray` form.

Loads the data provided from the disk for the file path given and returns the data as `numpy.ndarray` and name of the data sample. Names returned from this function will be used by the import cli system as the key for the returned data. This function can return either a single `numpy.ndarray`, sample name, combination, or a generator that produces one of the the above combinations. This helps when the input file is not a single data entry like an image but has multiple data points like CSV files.

An example implementation that returns a single data point:

```
def load(self, fpath, *args, **kwargs):
    data = create_np_array('myimg.jpg')
    name = create_sample_name('myimg.jpg')  # could use `self.sample_name`
    return data, name
```

An example implementation that returns a generator could look like this:

```
def load(self, fpath, *args, **kwargs):
    for i, line in enumerate('myfile.csv'):
        data = create_np_array(line)
        name = create_sample_name(fpath, i)
        yield data, name
```

static sample_name(*fpath: os.PathLike*) → str

Sample the name from file path.

This function comes handy since the *`load()`* method needs to `yield` or `return` both data and sample name. If there no specific requirements regarding sample name creation, you can use this function which removes the extension from the file name and returns just the name. For example, if filepath is `/path/to/myfile.ext`, then it returns `myfile`

Parameters **`fpath`** (*`os.PathLike`*) – Path to the file which is being loaded by *load*

`save` (*arr*, *outdir*, *sample_detail*, *extension*, **args*, ***kwargs*)

Save data in a `numpy.ndarray` to a specific file format on disk.

If the plugin is developed for files like CSV, JSON, etc - where multiple data entry would go to the same file - this should check whether the file exist already and weather it should modify / append the new data entry to the structure, instead of overwriting it or throwing an exception.

Note: Name of the file and the whole path to save the data should be constructed by this function. This can be done using the information gets as arguments such as, `outdir`, `sample_detail` and `extension`. It has been offloaded to this function instead of handling it before because, decisions like whether the multiple data entry should go to a single file or mutltpile file cannot be predicted before hand as are always data specific (and hence plugin specific)

Note: If the call to this function is initiated by the CLI, `sample_detail` argument will be a string formatted as *sample_name_type:sample_name*. For example, if the sample name is *sample1* (and type of sample name is *str*) then `sample_detail` will be *str:sample1*. This is to avoid the ambiguity that could arise by having both integer and string form of numerical as the sample name (ex: if column[123] and column["123"] exist). Formatting `sample_detail` to make a proper filename (not necessary) is upto the plugin developer.

`show` (*arr*, **args*, ***kwargs*)

Show/Display the data to the user.

This function should process the input `numpy.ndarray` and show that to the user using a data dependant display mechanism. A good example for such a system is `matplotlib.pyplot`'s `plt.show`, which displays the image data inline in the running terminal / kernel ui.

4.10 Frequently Asked Questions

The following documentation are taken from questions and comments on the Hangar User Group Slack Channel and over various Github issues.

4.10.1 How can I get an Invite to the Hangar User Group?

Just click on This Signup Link to get started.

4.10.2 Data Integrity

> Being a young project did you encounter some situations where the disaster was not a compilation error but dataset corruption? This is the most fearing aspect of using young projects but every project will start from a phase before becoming mature and production ready.

An absolute requirement of a system right this is to protect user data at all costs (I'll refer to this as preserving data "integrity" from here). During our initial design of the system, we made the decision that preserving integrity comes above all other system parameters: including performance, disk size, complexity of the Hangar core, and even features should we not be able to make them absolutely safe for the user. And to be honest, the very first versions of Hangar were quite slow and difficult to use as a result of this.

The initial versions of Hangar (which we put together in ~2 weeks) had essentially most of the features we have today. We've improved the API, made things clearer, and added some visualization/reporting utilities, but not much has changed. Essentially the entire development effort has been addressing issues stemming from a fundamental need to protect user data at all costs. That work has been very successful, and performance is extremely promising (and improving all the time).

To get into the details here: There have been only 3 instances in the entire time I've developed Hangar where we lost data irrecoverably:

1. We used to move data around between folders with some regularity (as a convenient way to mark some files as containing data which have been "committed", and can no longer be opened in anything but read-only mode). There was a bug (which never made it past a local dev version) at one point where I accidentally called `shutil.rmtree(path)` with a directory one level too high. . . that wasn't great.

   Just to be clear, we don't do this anymore (since disk IO costs are way too high), but remnants of it's intention are still very much alive and well. Once data has been added to the repository, and is "committed", the file containing that data will never be opened in anything but read-only mode again. This reduces the chance of disk corruption massively from the start.

2. When I was implementing the numpy memmap array storage backend, I was totally surprised during an early test when I:

   ```
   - opened a write-enabled checkout
   - added some data
   - without committing, retrieved the same data again via the user facing API
   - overwrote some slice of the return array with new data and did some processing
   - asked Hangar for that same array key again, and instead of returning
     the contents got a fatal RuntimeError raised by Hangar with the
     code/message indicating "'DATA CORRUPTION ERROR: Checksum {cksum} !=
     recorded for {hashVal}"
   ```

   What had happened was that when opening a `numpy.memmap` array on disk in `w+` mode, the default behavior when returning a subarray is to return a subclass of `np.ndarray` of type `np.memmap`. Though the numpy docs state: "The memmap object can be used anywhere an ndarray is accepted. Given a `memmap fp`, `isinstance(fp, numpy.ndarray)` returns `True`". I did not anticipate that updates to the subarray slice would also update the memmap on disk. A simple mistake to make; this has since been remedied by manually instantiating a new `np.ndarray` instance from the `np.memmap` subarray slice buffer.

   However, the nice part is that this was a real world proof that our system design worked (and not just in tests). When you add data to a Hangar checkout (or receive it on a fetch/clone operation) we calculate a hash digest of the data via `blake2b` (a cryptographically secure algorithm in the python standard library). While this allows us to cryptographically verify full integrity checks and history immutability, cryptographic hashes are slow by design. When we want to read local data (which we've already ensured was correct when it was placed on disk) it would be prohibitively slow to do a full cryptographic verification on every read. However, since its NOT acceptable to provide no integrity verification (even for local writes) we compromise with a much faster (though non cryptographic) hash digest/checksum. This operation occurs on EVERY read of data from disk.

The theory here is that even though Hangar makes every effort to guarantee safe operations itself, in the real world we have to deal with systems which break. We've planned for cases where some OS induced disk corruption occurs, or where some malicious actor modifies the file contents manually; we can't stop that from happening, but Hangar can make sure that you will know about it when it happens!

---

3. Before we got smart with the HDF5 backend low level details, it was an issue for us to have a write-enabled checkout attempt to write an array to disk and immediately read it back in. I'll gloss over the details for the sake of simplicity here, but basically I was presented with an CRC32 Checksum Verification Failed error in some edge cases. The interesting bit was that if I closed the checkout, and reopened it, it data was secure and intact on disk, but for immediate reads after writes, we weren't propagating changes to the HDF5 chunk metadata cache to `rw` operations appropriately.

   This was fixed very early on by taking advantage of a new feature in HDF5 1.10.4 referred to as Single Writer Multiple Reader (SWMR). The long and short is that by being careful to handle the order in which a new HDF5 file is created on disk and opened in w and r mode with SWMR enabled, the HDF5 core guarantees the integrity of the metadata chunk cache at all times. Even if a fatal system crash occurs in the middle of a write, the data will be preserved. This solved this issue completely for us

   There are many many many more details which I could cover here, but the long and short of it is that in order to ensure data integrity, Hangar is designed to not let the user do anything they aren't allowed to at any time

   - Read checkouts have no ability to modify contents on disk via any method. It's not possible for them to actually delete or overwrite anything in any way.
   - Write checkouts can only ever write data. The only way to remove the actual contents of written data from disk is if changes have been made in the staging area (but not committed) and the `reset_staging_area()` method is called. And even this has no ability to remove any data which had previously existed in some commit in the repo's history

   In addition, a Hangar checkout object is not what it appears to be (at first glance, use, or even during common introspection operations). If you try to operate on it after closing the checkout, or holding it while another checkout is started, you won't be able to (there's a whole lot of invisible "magic" going on with `weakrefs`, `objectproxies`, and instance attributes). I would encourage you to do the following:

```
>>> co = repo.checkout(write=True)
>>> co.metadata['hello'] = 'world'
>>> # try to hold a reference to the metadata object:
>>> mRef = co.metadata
>>> mRef['hello']
'world'
>>> co.commit('first commit')
>>> co.close()
>>> # what happens when you try to access the `co` or `mRef` object?
>>> mRef['hello']
ReferenceError: weakly-referenced object no longer exists
>>> print(co)  # or any other operation
PermissionError: Unable to operate on past checkout objects which have been␣
↪closed. No operation occurred. Please use a new checkout.
```

   The last bit I'll leave you with is a note on context managers and performance (how we handle record data safety and effectively

   See also:

   - *Hangar Tutorial* (Part 1, In section: "performance")

- *Hangar Under The Hood*

4.10.3 How Can a Hangar Repository be Backed Up?

Two strategies exist:

1. Use a remote server and Hangar's built in ability to just push data to a remote! (tutorial coming soon, see *Python API* for more details.
2. A Hangar repository is self contained in it's .hangar directory. To back up the data, just copy/paste or rsync it to another machine! (edited)

4.10.4 On Determining `Column` Schema Sizes

> Say I have a data group that specifies a data array with one dimension, three elements (say height, width, num channels) and later on I want to add bit depth. Can I do that, or do I need to make a new data group? Should it have been three scalar data groups from the start?

So right now it's not possible to change the schema (shape, dtype) of a column. I've thought about such a feature for a while now, and while it will require a new user facing API option, its (almost) trivial to make it work in the core. It just hasn't seemed like a priority yet. . .

And no, I wouldn't specify each of those as scalar data groups, they are a related piece of information, and generally would want to be accessed together

Access patterns should generally dictate how much info is placed in a column

Is there a performance/space penalty for having lots of small data groups?

As far as a performance / space penalty, this is where it gets good :)

- Using fewer columns means that there are fewer records (the internal locating info, kind-of like a git tree) to store, since each record points to a sample containing more information.
- Using more columns means that the likelihood of samples having the same value increases, meaning fewer pieces of data are actually stored on disk (remember it's a content addressable file store)

However, since the size of a record (40 bytes or so before compression, and we generally see compression ratios around 15-30% of the original size once the records are committed) is generally negligible compared to the size of data on disk, optimizing for number of records is just way overkill. For this case, it really doesn't matter. **Optimize for ease of use**

Note: The following documentation contains highly technical descriptions of the data writing and loading backends of the Hangar core. It is intended for developer use only, with the functionality described herein being completely hidden from regular users.

Any questions or comments can be directed to the Hangar Github Issues Page

4.11 Backend selection

Definition and dynamic routing to Hangar backend implementations.

This module defines the available backends for a Hangar installation & provides dynamic routing of method calls to the appropriate backend from a stored record specification.

4.11.1 Identification

A two character ascii code identifies which backend/version some record belongs to. Valid characters are the union of `ascii_lowercase`, `ascii_uppercase`, and `ascii_digits`:

```
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
```

Though stored as bytes in the backend, we use human readable characters (and not unprintable bytes) to aid in human tasks like developer database dumps and debugging. The characters making up the two digit code have the following symantic meanings:

- First Character (element 0) indicates the `backend type` used.
- Second character (element 1) indicates the `version` of the backend type which should be used to parse the specification & accesss data (more on this later)

The number of codes possible (a 2-choice permutation with repetition) is: 3844 which we anticipate to be more then sufficient long into the future. As a convention, the range of values in which the first digit of the code falls into can be used to identify the storage medium location:

- Lowercase `ascii_letters` & digits `[0, 1, 2, 3, 4]` -> reserved for backends handling data on the local disk.
- Uppercase `ascii_letters` & digits `[5, 6, 7, 8, 9]` -> reserved for backends referring to data residing on a remote server.

This is not a hard and fast rule though, and can be changed in the future if the need arises.

4.11.2 Process & Guarantees

In order to maintain backwards compatibility across versions of Hangar into the future the following ruleset is specified and MUST BE HONORED:

- When a new backend is proposed, the contributor(s) provide the class with a meaningful name (`HDF5`, `NUMPY`, `TILEDB`, etc) identifying the backend to Hangar developers. The review team will provide:
  - `backend type` code
  - `version` code

  which all records related to that implementation identify themselves with. In addition, Externally facing classes / methods go by a canonical name which is the concatenation of the `meaningful name` and assigned `"format code"` ie. for `backend name: 'NUMPY'` assigned `type code: '1'` and `version code: '0'` must start external method/class names with: `NUMPY_10_foo`
- Once a new backend is accepted, the code assigned to it is PERMANENT & UNCHANGING. The same code cannot be used in the future for other backends.
- Each backend independently determines the information it needs to log/store to uniquely identify and retrieve a sample stored by it. There is no standard format, each is free to define whatever fields they find most convenient. Unique encode/decode methods are defined in order to serialize this information to bytes and then reconstruct

the information later. These bytes are what are passed in when a retrieval request is made, and returned when a storage request for some piece of data is performed.

- Once accepted, The record format specified (ie. the byte representation described above) cannot be modified in any way. This must remain permanent!
- Backend (internal) methods can be updated, optimized, and/or changed at any time so long as:
  - No changes to the record format specification are introduced
  - Data stored via any previous iteration of the backend's accessor methods can be retrieved bitwise exactly by the "updated" version.

Before proposing a new backend or making changes to this file, please consider reaching out to the Hangar core development team so we can guide you through the process.

4.11.3 Backend Specifications

Local HDF5 Backend

Local HDF5 Backend Implementation, Identifier: `HDF5_00`

Backend Identifiers

- Backend: `0`
- Version: `0`
- Format Code: `00`
- Canonical Name: `HDF5_00`

Storage Method

- Data is written to specific subarray indexes inside an HDF5 "dataset" in a single HDF5 File.
- In each HDF5 File there are `COLLECTION_COUNT` "datasets" (named `["0" : "{COLLECTION_COUNT}"]`). These are referred to as `"dataset number"`
- Each dataset is a zero-initialized array of:
  - `dtype: {schema_dtype}`; ie `np.float32` or `np.uint8`
  - `shape: (COLLECTION_SIZE, *{schema_shape.size})`; ie `(500, 10)` or `(500, 300)`. The first index in the dataset is referred to as a `collection index`. See technical note below for detailed explanation on why the flatten operaiton is performed.
- Compression Filters, Chunking Configuration/Options are applied globally for all `datasets` in a file at dataset creation time.
- On read and write of all samples the xxhash64_hexdigest is calculated for the raw array bytes. This is to ensure that all data in == data out of the hdf5 files. That way even if a file is manually edited (bypassing fletcher32 filter check) we have a quick way to tell that things are not as they should be.

Compression Options

Accepts dictionary containing keys

- `backend == "00"`
- `complib`
- `complevel`
- `shuffle`

Blosc-HDF5

- `complib` valid values:
  - `'blosc:blosclz'`,
  - `'blosc:lz4'`,
  - `'blosc:lz4hc'`,
  - `'blosc:zlib'`,
  - `'blosc:zstd'`
- `complevel` valid values: [0, 9] where 0 is "no compression" and 9 is "most compression"
- `shuffle` valid values:
  - `None`
  - `'none'`
  - `'byte'`
  - `'bit'`

LZF Filter

- `'complib' == 'lzf'`
- `'shuffle'` one of `[False, None, 'none', True, 'byte']`
- `'complevel'` one of `[False, None, 'none']`

GZip Filter

- `'complib' == 'gzip'`
- `'shuffle'` one of `[False, None, 'none', True, 'byte']`
- `complevel` valid values: [0, 9] where 0 is "no compression" and 9 is "most compression"

Record Format

Fields Recorded for Each Array

- Format Code
- File UID
- xxhash64_hexdigest (ie. checksum)
- Dataset Number (`0:COLLECTION_COUNT` dataset selection)
- Dataset Index (`0:COLLECTION_SIZE` dataset subarray selection)

- Subarray Shape

Examples

1) Adding the first piece of data to a file:

    - Array shape (Subarray Shape): (10, 10)
    - File UID: "rlUK3C"
    - xxhash64_hexdigest: 8067007c0f05c359
    - Dataset Number: 16
    - Collection Index: 105

    ```
    Record Data => "00:rlUK3C:8067007c0f05c359:16:105:10 10"
    ```

1) Adding to a piece of data to a the middle of a file:

    - Array shape (Subarray Shape): (20, 2, 3)
    - File UID: "rlUK3C"
    - xxhash64_hexdigest: b89f873d3d153a9c
    - Dataset Number: "3"
    - Collection Index: 199

    ```
    Record Data => "00:rlUK3C:b89f873d3d153a9c:8:199:20 2 3"
    ```

Technical Notes

- Files are read only after initial creation/writes. Only a write-enabled checkout can open a HDF5 file in `"w"` or `"a"` mode, and writer checkouts create new files on every checkout, and make no attempt to fill in unset locations in previous files. This is not an issue as no disk space is used until data is written to the initially created "zero-initialized" collection datasets
- On write: Single Writer Multiple Reader (`SWMR`) mode is set to ensure that improper closing (not calling `.close()`) method does not corrupt any data which had been previously flushed to the file.
- On read: SWMR is set to allow multiple readers (in different threads / processes) to read from the same file. File handle serialization is handled via custom python `pickle` serialization/reduction logic which is implemented by the high level `pickle` reduction `__set_state__()`, `__get_state__()` class methods.
- An optimization is performed in order to increase the read / write performance of variable shaped datasets. Due to the way that we initialize an entire HDF5 file with all datasets pre-created (to the size of the max subarray shape), we need to ensure that storing smaller sized arrays (in a variable sized Hangar Column) would be effective. Because we use chunked storage, certain dimensions which are incomplete could have potentially required writes to chunks which do are primarily empty (worst case "C" index ordering), increasing read / write speeds significantly.

  To overcome this, we create HDF5 datasets which have `COLLECTION_SIZE` first dimension size, and only ONE second dimension of size `schema_shape.size()` (ie. product of all dimensions). For example an array schema with shape (10, 10, 3) would be stored in a HDF5 dataset of shape (COLLECTION_SIZE, 300). Chunk sizes are chosen to align on the first dimension with a second dimension of size which fits the total data into L2 CPU Cache (< 256 KB). On write, we use the `np.ravel` function to construct a "view" (not copy) of the array as a 1D array, and then on read we reshape the array to the recorded size (a copyless "view-only" operation). This is part of the reason that we only accept C ordered arrays as input to Hangar.

Fixed Shape Optimized Local HDF5

Local HDF5 Backend Implementation, Identifier: `HDF5_01`

Backend Identifiers

- Backend: `0`
- Version: `1`
- Format Code: `01`
- Canonical Name: `HDF5_01`

Storage Method

- This module is meant to handle larger datasets which are of fixed size. IO and significant compression optimization is achieved by storing arrays at their appropriate top level index in the same shape they naturally assume and chunking over the entire subarray domain making up a sample (rather than having to subdivide chunks when the sample could be variably shaped.)
- Data is written to specific subarray indexes inside an HDF5 "dataset" in a single HDF5 File.
- In each HDF5 File there are `COLLECTION_COUNT` "datasets" (named `["0" : "{COLLECTION_COUNT}"]`). These are referred to as `"dataset number"`
- Each dataset is a zero-initialized array of:
  - `dtype: {schema_dtype}`; ie `np.float32` or `np.uint8`
  - `shape: (COLLECTION_SIZE, *{schema_shape})`; ie `(500, 10, 10)` or `(500, 512, 512, 320)`. The first index in the dataset is referred to as a `collection index`.
- Compression Filters, Chunking Configuration/Options are applied globally for all `datasets` in a file at dataset creation time.
- On read and write of all samples the xxhash64_hexdigest is calculated for the raw array bytes. This is to ensure that all data in == data out of the hdf5 files. That way even if a file is manually edited (bypassing fletcher32 filter check) we have a quick way to tell that things are not as they should be.

Compression Options

Accepts dictionary containing keys

- `backend` == `"01"`
- `complib`
- `complevel`
- `shuffle`

Blosc-HDF5

- `complib` valid values:
  - `'blosc:blosclz'`,
  - `'blosc:lz4'`,

  - 'blosc:lz4hc',
  - 'blosc:zlib',
  - 'blosc:zstd'
- complevel valid values: [0, 9] where 0 is “no compression” and 9 is “most compression”
- shuffle valid values:
  - None
  - 'none'
  - 'byte'
  - 'bit'

LZF Filter

- 'complib' == 'lzf'
- 'shuffle' one of [False, None, 'none', True, 'byte']
- 'complevel' one of [False, None, 'none']

GZip Filter

- 'complib' == 'gzip'
- 'shuffle' one of [False, None, 'none', True, 'byte']
- complevel valid values: [0, 9] where 0 is “no compression” and 9 is “most compression”

Record Format

Fields Recorded for Each Array

- Format Code
- File UID
- xxhash64_hexdigest (ie. checksum)
- Dataset Number (0:COLLECTION_COUNT dataset selection)
- Dataset Index (0:COLLECTION_SIZE dataset subarray selection)
- Subarray Shape

Examples

1) Adding the first piece of data to a file:
   - Array shape (Subarray Shape): (10, 10)
   - File UID: “rlUK3C”
   - xxhash64_hexdigest: 8067007c0f05c359
   - Dataset Number: 16
   - Collection Index: 105

   ```
   Record Data => "01:rlUK3C:8067007c0f05c359:16:105:10 10"
   ```

1) Adding to a piece of data to a the middle of a file:

   - Array shape (Subarray Shape): (20, 2, 3)
   - File UID: “rlUK3C”
   - xxhash64_hexdigest: b89f873d3d153a9c
   - Dataset Number: “3”
   - Collection Index: 199

   ```
   Record Data => "01:rlUK3C:b89f873d3d153a9c:8:199:20 2 3"
   ```

Technical Notes

- The majority of methods not directly related to “chunking” and the “raw data chunk cache” are either identical to HDF5_00, or only slightly modified.
- Files are read only after initial creation/writes. Only a write-enabled checkout can open a HDF5 file in `"w"` or `"a"` mode, and writer checkouts create new files on every checkout, and make no attempt to fill in unset locations in previous files. This is not an issue as no disk space is used until data is written to the initially created “zero-initialized” collection datasets
- On write: Single Writer Multiple Reader (`SWMR`) mode is set to ensure that improper closing (not calling `.close()`) method does not corrupt any data which had been previously flushed to the file.
- On read: SWMR is set to allow multiple readers (in different threads / processes) to read from the same file. File handle serialization is handled via custom python `pickle` serialization/reduction logic which is implemented by the high level `pickle` reduction `__set_state__()`, `__get_state__()` class methods.
- An optimization is performed in order to increase the read / write performance of fixed size datasets. Due to the way that we initialize an entire HDF5 file with all datasets pre-created (to the size of the fixed subarray shape), and the fact we absolutely know the size / shape / access-pattern of the arrays, inefficient IO due to wasted chunk processing is not a concern. It is far more efficient for us to completely blow off the metadata chunk cache, and chunk each subarray as a single large item item.

  This method of processing tends to have a number of significant effects as compared to chunked storage methods:

  1. **Compression rations improve** (by a non-trivial factor). This is simply due to the fact that a larger amount of raw data is being passed into the compressor at a time. While the exact improvement seen is highly dependent on both the data size and compressor used, there should be no case where compressing the full tensor uses more disk space then chunking the tensor, compressing each chunk individually, and then saving each chunk to disk.
  2. **Read performance improves** (so long as a suitable compressor / option set was chosen). Instead of issuing (potentially) many read requests - one for each chunk - to the storage hardware, signifiantly few IOPS are used to retrieve the entire set of compressed raw data from disk. Fewer IOPS means much less time waiting on the hard disk. Moreso, only a single decompression step is needed to reconstruct the numeric array, completly decoupling performance from HDF5’s ability to parallelize internal filter pipeline operations.

     Additionally, since the entire requested chunk is retrieved in a single decompression pipeline run, there is no need for the HDF5 core to initialize an intermediate buffer which holds data chunks as each decompression operation completes. Futher, by preinitializing an empty `numpy.ndarray` container and using the low level HDF5 `read_direct` method, the decompressed data buffer is passes directly into the returned `ndarray.__array_interface__.data` field with no intermediate copy or processing steps.
  3. **Shuffle filters are favored.**. With much more data to work with in a single compression operation, the use of “byte shuffle” filters in the compressor spec has been seen to both markedly decrease read time and

increase compression ratios. Shuffling can significantly reduce disk space required to store some piece of data on disk, further reducing the time spent waiting on hard disk IO while incuring a negligible cost to decompression speed.

Taking all of these effects into account, there can be up to an order of magnitude increase in read performance as compared to the subarray chunking strategy employed by the `HDF5_00` backend.

- Like all other backends at the time of writing, only 'C' ordered arrays are accepted by this method.

Local NP Memmap Backend

Local Numpy memmap Backend Implementation, Identifier: `NUMPY_10`

Backend Identifiers

- Backend: `1`
- Version: `0`
- Format Code: `10`
- Canonical Name: `NUMPY_10`

Storage Method

- Data is written to specific subarray indexes inside a numpy memmapped array on disk.
- Each file is a zero-initialized array of
  - `dtype: {schema_dtype}`; ie `np.float32` or `np.uint8`
  - `shape: (COLLECTION_SIZE, *{schema_shape})`; ie `(500, 10)` or `(500, 4, 3)`. The first index in the array is referred to as a "collection index".

Compression Options

Does not accept any compression options. No compression is applied.

Record Format

Fields Recorded for Each Array

- Format Code
- File UID
- xxhash64_hexdigest
- Collection Index (0:COLLECTION_SIZE subarray selection)
- Subarray Shape

Examples

1) Adding the first piece of data to a file:

   - Array shape (Subarray Shape): (10, 10)
   - File UID: "K3ktxv"
   - xxhash64_hexdigest: 94701dd9f32626e2
   - Collection Index: 488

   ```
   Record Data => "10:K3ktxv:94701dd9f32626e2:488:10 10"
   ```

2) Adding to a piece of data to a the middle of a file:

   - Array shape (Subarray Shape): (20, 2, 3)
   - File UID: "Mk23nl"
   - xxhash64_hexdigest: 1363344b6c051b29
   - Collection Index: 199

   ```
   Record Data => "10:Mk23nl:1363344b6c051b29:199:20 2 3"
   ```

Technical Notes

- A typical numpy memmap file persisted to disk does not retain information about its datatype or shape, and as such must be provided when re-opened after close. In order to persist a memmap in `.npy` format, we use the a special function `open_memmap` imported from `np.lib.format` which can open a memmap file and persist necessary header info to disk in `.npy` format.
- On each write, an `xxhash64_hexdigest` checksum is calculated. This is not for use as the primary hash algorithm, but rather stored in the local record format itself to serve as a quick way to verify no disk corruption occurred. This is required since numpy has no built in data integrity validation methods when reading from disk.

Variable Shape LMDB String Data Store

Local LMDB Backend Implementation, Identifier: `LMDB_30`

Backend Identifiers

- Backend: `3`
- Version: `0`
- Format Code: `30`
- Canonical Name: `LMDB_30`

Storage Method

- This module is meant to handle string typed data which is of any size. IO is performed via the LMDB storage system.
- This module does not compress values upon writing, the full (uncompressed) value of the text is written to the DB for each key.
- For each LMDB file generated, data is indexed by keys which are generated in lexicographically sorted order of key length 4. Keys consist of 4 characters chosen from an alphabet consisting of ASCII digits, lowercase letters, and upercase letters. Within a single write instance (when an LMDB file is created and written to), lexicographically sorted permutations of the chosen characters are used as key indexes.

  This means that for each LMDB file written in a repo, the sequence of generated index keys will be identical, even though two databases with the same key will store different values. As such, the File UID is crucial in order to identify a unique db/index key combo to access a particular value by.
- There is no limit to the size which each record can occupy. Data is stored "as-is" and is uncompressed. Reading the data back will return the exact data stored (regardless of how large the data record is).
- On read and write of all samples the xxhash64_hexdigest is calculated for the raw data bytes. This is to ensure that all data in == data out of the lmdb files. That way even if a file is manually edited we have a quick way to tell that things are not as they should be. (full data hash digests may not be calculated every time a read is performed).

Compression Options

None

Record Format

Fields Recorded for Each Array

- Format Code
- File UID
- Row Index

Examples

1) Adding the first piece of data to a file:

   - File UID: "rlUK3C"
   - Row Index: "0123"
   - xxhash64_hexdigest: 8067007c0f05c359

   ```
   Record Data => "30:rlUK3C:0123:8067007c0f05c359"
   ```

2) Adding a second piece of data:

   - File UID: "rlUK3C"
   - Row Index: "0124"
   - xxhash64_hexdigest: b89f873d3d153a9c

```
Record Data => "30:rlUK3C:0124:b89f873d3d153a9c"
```

3) Adding a the 500th piece of data:
   - File UID: “rlUK3C”
   - Row Index: “01AU”
   - xxhash64_hexdigest: cf3fc53cad153a5a

```
Record Data => "30:rlUK3C:01AU:cf3fc53cad153a5a"
```

Remote Server Unknown Backend

Remote server location unknown backend, Identifier: `REMOTE_50`

Backend Identifiers

- Backend: `5`
- Version: `0`
- Format Code: `50`
- Canonical Name: `REMOTE_50`

Storage Method

- This backend merely acts to record that there is some data sample with some `hash` and `schema_shape` present in the repository. It does not store the actual data on the local disk, but indicates that if it should be retrieved, you need to ask the remote hangar server for it. Once present on the local disk, the backend locating info will be updated with one of the *local* data backend specifications.

Record Format

Fields Recorded for Each Array

- Format Code
- Schema Hash

Separators used

- `SEP_KEY: ":"`

Examples

1) Adding the first piece of data to a file:

   - Schema Hash: “ae43A21a”

   ```
   Record Data => '50:ae43A21a'
   ```

1) Adding to a piece of data to a the middle of a file:

   - Schema Hash: “ae43A21a”

   ```
   Record Data => '50:ae43A21a'
   ```

Technical Notes

- The schema_hash field is required in order to allow effective placement of actual retrieved data into suitable sized collections on a `fetch-data()` operation

4.12 Contributing to Hangar

4.12.1 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

All community members should read and abide by our *Contributor Code of Conduct*.

Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Documentation improvements

Hangar could always use more documentation, whether as part of the official Hangar docs, in docstrings, or even on the web in blog posts, articles, and such.

Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/tensorwerk/hangar-py/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

Development

To set up *hangar-py* for local development:

1. Fork hangar-py (look for the “Fork” button).
2. Clone your fork locally:

   ```
   git clone git@github.com:your_name_here/hangar-py.git
   ```

3. Create a branch for local development:

   ```
   git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

4. When you’re done making changes, run all the checks, doc builder and spell checker with tox one command:

   ```
   tox
   ```

5. Commit your changes and push your branch to GitHub:

   ```
   git add .
   git commit -m "Your detailed description of your changes."
   git push origin name-of-your-bugfix-or-feature
   ```

6. Submit a pull request through the GitHub website.

Pull Request Guidelines

If you need some code review or feedback while you’re developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there’s new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don’t have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.

It will be slower though . . .

4.12.2 Contributor Code of Conduct

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at hangar.info@tensorwerk.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

Attribution

This Code of Conduct is adapted from the Contributor Covenant homepage, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

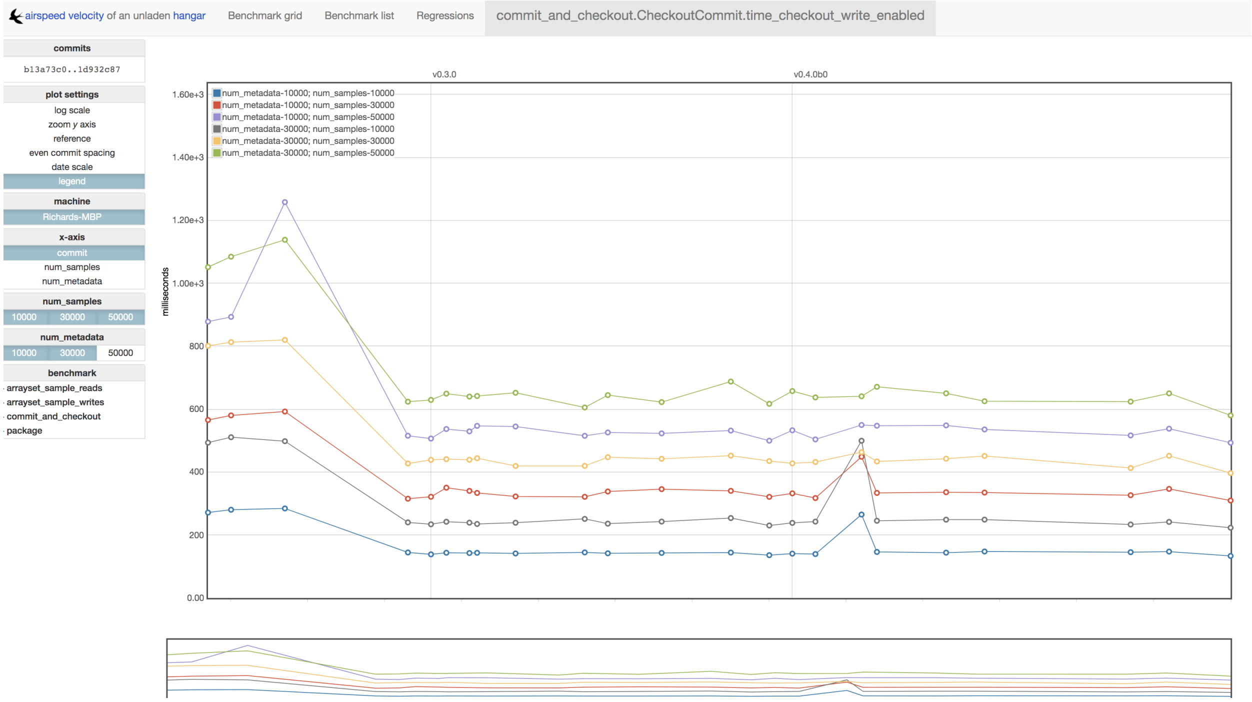
For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq

4.12.3 Hangar Performance Benchmarking Suite

A set of benchmarking tools are included in order to track the performance of common hangar operations over the course of time. The benchmark suite is run via the phenomenal Airspeed Velocity (ASV) project.

Benchmarks can be viewed at the following web link, or by examining the raw data files in the separate benchmark results repo.

- Benchmark Web View
- Benchmark Results Repo



Purpose

In addition to providing historical metrics and insight into application performance over many releases of Hangar, *the benchmark suite is used as a canary to identify potentially problematic pull requests.* All PRs to the Hangar repository are automatically benchmarked by our CI system to compare the performance of proposed changes to that of the current `master` branch.

The results of this canary are explicitly NOT to be used as the "be-all-end-all" decider of whether a PR is suitable to be merged or not.

Instead, it is meant to serve the following purposes:

1. **Help contributors understand the consequences of some set of changes on the greater system early in the PR process.** Simple code is best; if there's no obvious performance degradation or significant improvement to be had, then there's no need (or really rationale) for using more complex algorithms or data structures. It's more work for the author, project maintainers, and long term health of the codebase.
2. **Not everything can be caught by the capabilities of a traditional test suite.** Hangar is fairly flat/modular in structure, but there are certain hotspots in the codebase where a simple change could drastically degrade performance. It's not always obvious where these hotspots are, and even a change which is functionally identical (introducing no issues/bugs to the end user) can unknowingly cross a line and introduce some large regression completely unnoticed to the authors/reviewers.
3. Sometimes tradeoffs need to be made when introducing something new to a system. Whether this be due to fundamental CS problems (space vs. time) or simple matters of practicality vs. purity, it's always easier to act in environments where relevant information is available before a decision is made. **Identifying and quantifying tradeoffs/regressions/benefits during development is the only way we can make informed decisions.** The only times to be OK with some regression is when knowing about it in advance, it might be the right choice at the time, but if we don't measure we will never know.

Important Notes on Using/Modifying the Benchmark Suite

1. **Do not commit any of the benchmark results, environment files, or generated visualizations to the repository**. We store benchmark results in a separate repository so to not clutter the main repo with un-necessary data. The default directories these are generated in are excluded in our `.gitignore` config, so baring some unusual git usage patterns, this should not be a day-to-day concern.
2. Proposed changes to the benchmark suite should be made to the code in this repository first. The benchmark results repository mirror will be synchronized upon approval/merge of changes to the main Hangar repo.

Introduction to Running Benchmarks

As ASV sets up and manages it's own virtual environments and source installations, benchmark execution is not run via `tox`. While a brief tutorial is included below, please refer to the ASV Docs for detailed information on how to both run, understand, and write ASV benchmarks.

First Time Setup

1. Ensure that `virtualenv`, `setuptools`, `pip` are updated to the latest version.
2. Install ASV `$ pip install asv`.
3. Open a terminal and navigate to the `hangar-py/asv-bench` directory.
4. Run `$ asv machine` to record details of your machine, it is OK to just use the defaults.

Running Benchmarks

Refer to the using ASV page for a full tutorial, paying close attention to the asv run command. Generally `asv run` requires a range of commits to benchmark across (specified via either branch name, tags, or commit digests).

To benchmark every commit between the current master `HEAD` and `v0.3.0`, you would execute:

```
$ asv run v0.2.0..master
```

However, this may result in a larger workload then you are willing to wait around for. To limit the number of commits, you can specify the `--steps=N` option to only benchmark `N` commits at most between `HEAD` and `v0.3.0`.

The most useful tool during development is the asv continuous command. using the following syntax will benchmark any changes in a local development branch against the base `master` commit:

```
$ asv continuous origin/master HEAD
```

Running asv compare will generate a quick summary of any performance differences:

```
$ asv compare origin/master HEAD
```

Visualizing Results

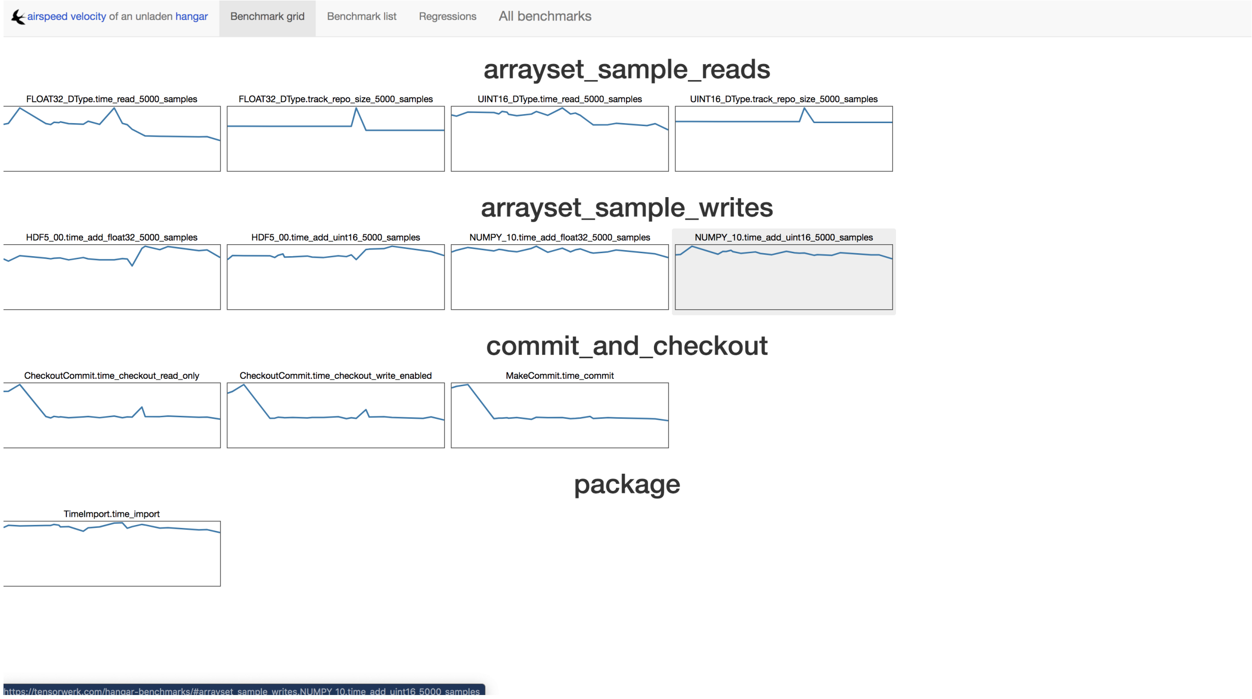
After generating benchmark data for a number of commits through history, the results can be reviewed in (an automatically generated) local web interface by running the following commands:

```
$ asv publish
$ asv preview
```

Navigating to `http://127.0.0.1:8080/` will pull up an interactive webpage where the full set of benchmark graphs/explorations utilities can be viewed. This will look something like the image below.

4.13 Authors

- Richard Izzo - rick@tensorwerk.com
- Luca Antiga - luca@tensorwerk.com
- Sherin Thomas - sherin@tensorwerk.com
- Alessia Marcolini - alessia@tensorwerk.com


airspeed velocity of an unladen hangar
Benchmark grid
Benchmark list
Regressions
All benchmarks
arrayset_sample_reads
FLOAT32_DType.time_read_5000_samples
FLOAT32_DType.track_repo_size_5000_samples
UINT16_DType.time_read_5000_samples
UINT16_DType.track_repo_size_5000_samples
arrayset_sample_writes
HDF5_00.time_add_float32_5000_samples
HDF5_00.time_add_uint16_5000_samples
NUMPY_10.time_add_float32_5000_samples
NUMPY_10.time_add_uint16_5000_samples
commit_and_checkout
CheckoutCommit.time_checkout_read_only
CheckoutCommit.time_checkout_write_enabled
MakeCommit.time_commit
package
TimeImport.time_import
https://tensorwerk.com/hangar-benchmarks/#arrayset_sample_writes.NUMPY_10.time_add_uint16_5000_samples

4.14 Change Log

4.14.1 v0.5.2 (2020-05-08)

New Features

- New column data type supporting arbitrary `bytes` data. (#198) @rlizzo

Improvements

- `str` typed columns can now accept data containing any unicode code-point. In prior releases data containing any `non-ascii` character could not be written to this column type. (#198) @rlizzo

Bug Fixes

- Fixed issue where `str` and (newly added) `bytes` column data could not be fetched / pushed between a local client repository and remote server. (#198) @rlizzo

4.14.2 v0.5.1 (2020-04-05)

BugFixes

- Fixed issue where importing `make_torch_dataloader` or `make_tf_dataloader` under python 3.6 Would raise a `NameError` irrigardless of if the package is installed. (#196) @rlizzo

4.14.3 v0.5.0 (2020-04-4)

Improvements

- Python 3.8 is now fully supported. (#193) @rlizzo
- Major backend overhaul which defines column layouts and data types in the same interchangable / extensable manner as storage backends. This will allow rapid development of new layouts and data type support as new use cases are discovered by the community. (#184) @rlizzo
- Column and backend classes are now fully serializable (pickleable) for `read-only` checkouts. (#180) @rlizzo
- Modularized internal structure of API classes to easily allow new columnn layouts / data types to be added in the future. (#180) @rlizzo
- Improved type / value checking of manual specification for column `backend` and `backend_options`. (#180) @rlizzo
- Standardized column data access API to follow python standard library `dict` methods API. (#180) @rlizzo

- Memory usage of arrayset checkouts has been reduced by ~70% by using C-structs for allocating sample record locating info. (#179) @rlizzo
- Read times from the `HDF5_00` and `HDF5_01` backend have been reduced by 33-38% (or more for arraysets with many samples) by eliminating redundant computation of chunked storage B-Tree. (#179) @rlizzo
- Commit times and checkout times have been reduced by 11-18% by optimizing record parsing and memory allocation. (#179) @rlizzo

New Features

- Added `str` type column with same behavior as `ndarray` column (supporting both single-level and nested layouts) added to replace functionality of removed `metadata` container. (#184) @rlizzo
- New backend based on `LMDB` has been added (specifier of `lmdb_30`). (#184) @rlizzo
- Added `.diff()` method to `Repository` class to enable diffing changes between any pair of commits / branches without needing to open the diff base in a checkout. (#183) @rlizzo
- New CLI command `hangar diff` which reports a summary view of changes made between any pair of commits / branches. (#183) @rlizzo
- Added `.log()` method to `Checkout` objects so graphical commit graph or machine readable commit details / DAG can be queried when operating on a particular commit. (#183) @rlizzo
- "string" type columns now supported alongside "ndarray" column type. (#180) @rlizzo
- New "column" API, which replaces "arrayset" name. (#180) @rlizzo
- Arraysets can now contain "nested subsamples" under a common sample key. (#179) @rlizzo
- New API to add and remove samples from and arrayset. (#179) @rlizzo
- Added `repo.size_nbytes` and `repo.size_human` to report disk usage of a repository on disk. (#174) @rlizzo
- Added method to traverse the entire repository history and cryptographically verify integrity. (#173) @rlizzo

Changes

- Argument syntax of `__getitem__()` and `get()` methods of `ReaderCheckout` and `WriterCheckout` classes. The new format supports handeling arbitrary arguments specific to retrieval of data from any column type. (#183) @rlizzo

Removed

- `metadata` container for `str` typed data has been completly removed. It is replaced by a highly extensible and much more user-friendly `str` typed column. (#184) @rlizzo
- `__setitem__()` method in `WriterCheckout` objects. Writing data to columns via a checkout object is no longer supported. (#183) @rlizzo

Bug Fixes

- Backend data stores no longer use file symlinks, improving compatibility with some types file systems. (#171) @rlizzo
- All arrayset types ("flat" and "nested subsamples") and backend readers can now be pickled – for parallel processing – in a read-only checkout. (#179) @rlizzo

Breaking changes

- New backend record serialization format is incompatible with repositories written in version 0.4 or earlier.
- New arrayset API is incompatible with Hangar API in version 0.4 or earlier.

4.14.4 v0.4.0 (2019-11-21)

New Features

- Added ability to delete branch names/pointers from a local repository via both API and CLI. (#128) @rlizzo
- Added `local` keyword arg to arrayset key/value iterators to return only locally available samples (#131) @rlizzo
- Ability to change the backend storage format and options applied to an `arrayset` after initialization. (#133) @rlizzo
- Added blosc compression to HDF5 backend by default on PyPi installations. (#146) @rlizzo
- Added Benchmarking Suite to Test for Performance Regressions in PRs. (#155) @rlizzo
- Added new backend optimized to increase speeds for fixed size arrayset access. (#160) @rlizzo

Improvements

- Removed `msgpack` and `pyyaml` dependencies. Cleaned up and improved remote client/server code. (#130) @rlizzo
- Multiprocess Torch DataLoaders allowed on Linux and MacOS. (#144) @rlizzo
- Added CLI options `commit`, `checkout`, `arrayset create`, & `arrayset remove`. (#150) @rlizzo
- Plugin system revamp. (#134) @hhsecond
- Documentation Improvements and Typo-Fixes. (#156) @alessiamarcolini
- Removed implicit removal of arrayset schema from checkout if every sample was removed from arrayset. This could potentially result in dangling accessors which may or may not self-destruct (as expected) in certain edge-cases. (#159) @rlizzo
- Added type codes to hash digests so that calculation function can be updated in the future without breaking repos written in previous Hangar versions. (#165) @rlizzo

Bug Fixes

- Programatic access to repository log contents now returns branch heads alongside other log info. (#125) @rlizzo
- Fixed minor bug in types of values allowed for `Arrayset` names vs `Sample` names. (#151) @rlizzo
- Fixed issue where using checkout object to access a sample in multiple arraysets would try to create a `namedtuple` instance with invalid field names. Now incompatible field names are automatically renamed with their positional index. (#161) @rlizzo
- Explicitly raise error if `commit` argument is set while checking out a repository with `write=True`. (#166) @rlizzo

Breaking changes

- New commit reference serialization format is incompatible with repositories written in version 0.3.0 or earlier.

4.14.5 v0.3.0 (2019-09-10)

New Features

- API addition allowing reading and writing arrayset data from a checkout object directly. (#115) @rlizzo
- Data importer, exporters, and viewers via CLI for common file formats. Includes plugin system for easy extensibility in the future. (#103) (@rlizzo, @hhsecond)

Improvements

- Added tutorial on working with remote data. (#113) @rlizzo
- Added Tutorial on Tensorflow and PyTorch Dataloaders. (#117) @hhsecond
- Large performance improvement to diff/merge algorithm (~30x previous). (#112) @rlizzo
- New commit hash algorithm which is much more reproducible in the long term. (#120) @rlizzo
- HDF5 backend updated to increase speed of reading/writing variable sized dataset compressed chunks (#120) @rlizzo

Bug Fixes

- Fixed ML Dataloaders errors for a number of edge cases surrounding partial-remote data and non-common keys. (#110) (@hhsecond, @rlizzo)

Breaking changes

- New commit hash algorithm is incompatible with repositories written in version 0.2.0 or earlier

4.14.6 v0.2.0 (2019-08-09)

New Features

- Numpy memory-mapped array file backend added. (#70) @rlizzo
- Remote server data backend added. (#70) @rlizzo
- Selection heuristics to determine appropriate backend from arrayset schema. (#70) @rlizzo
- Partial remote clones and fetch operations now fully supported. (#85) @rlizzo
- CLI has been placed under test coverage, added interface usage to docs. (#85) @rlizzo
- TensorFlow and PyTorch Machine Learning Dataloader Methods (*Experimental Release*). (#91) lead: @hhsecond, co-author: @rlizzo, reviewed by: @elistevens

Improvements

- Record format versioning and standardization so to not break backwards compatibility in the future. (#70) @rlizzo
- Backend addition and update developer protocols and documentation. (#70) @rlizzo
- Read-only checkout arrayset sample `get` methods now are multithread and multiprocess safe. (#84) @rlizzo
- Read-only checkout metadata sample `get` methods are thread safe if used within a context manager. (#101) @rlizzo
- Samples can be assigned integer names in addition to `string` names. (#89) @rlizzo
- Forgetting to close a `write-enabled` checkout before terminating the python process will close the checkout automatically for many situations. (#101) @rlizzo
- Repository software version compatability methods added to ensure upgrade paths in the future. (#101) @rlizzo
- Many tests added (including support for Mac OSX on Travis-CI). lead: @rlizzo, co-author: @hhsecond

Bug Fixes

- Diff results for fast forward merges now returns sensible results. (#77) @rlizzo
- Many type annotations added, and developer documentation improved. @hhsecond & @rlizzo

Breaking changes

- Renamed all references to `datasets` in the API / world-view to `arraysets`.
- These are backwards incompatible changes. For all versions > 0.2, repository upgrade utilities will be provided if breaking changes occur.

4.14.7 v0.1.1 (2019-05-24)

Bug Fixes

- Fixed typo in README which was uploaded to PyPi

4.14.8 v0.1.0 (2019-05-24)

New Features

- Remote client-server config negotiation and administrator permissions. (#10) @rlizzo
- Allow single python process to access multiple repositories simultaneously. (#20) @rlizzo
- Fast-Forward and 3-Way Merge and Diff methods now fully supported and behaving as expected. (#32) @rlizzo

Improvements

- Initial test-case specification. (#14) @hhsecond
- Checkout test-case work. (#25) @hhsecond
- Metadata test-case work. (#27) @hhsecond
- Any potential failure cases raise exceptions instead of silently returning. (#16) @rlizzo
- Many usability improvements in a variety of commits.

Bug Fixes

- Ensure references to checkout arrayset or metadata objects cannot operate after the checkout is closed. (#41) @rlizzo
- Sensible exception classes and error messages raised on a variety of situations (Many commits). @hhsecond & @rlizzo
- Many minor issues addressed.

API Additions

- Refer to API documentation (#23)

Breaking changes

- All repositories written with previous versions of Hangar are liable to break when using this version. Please upgrade versions immediately.

4.14.9 v0.0.0 (2019-04-15)

- First Public Release of Hangar!

CHAPTER FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX