Hangar

Release 0.2.0

Oct 08, 2019

Contents

1 What is Hangar? 3

2 Installation 5

3 Documentation 7

4 Development 9

4.1 Overview . . . 9
4.1.1 What is Hangar? . . . 10
4.1.2 Installation . . . 10
4.1.3 Documentation . . . 11
4.1.4 Development . . . 11
4.2 Usage . . . 11
4.3 Installation . . . 11
4.3.1 Pre-Built Installation . . . 11
4.3.2 Source Installation . . . 12
4.4 Hangar Core Concepts . . . 12
4.4.1 What Is Hangar? . . . 12
4.4.2 Inspiration . . . 12
4.4.3 How Hangar Thinks About Data . . . 13
4.4.4 Implications of the Hangar Data Philosophy . . . 15
4.4.5 What's Next? . . . 20
4.5 Hangar Tutorial . . . 20
4.5.1 Part 1: Creating A Repository And Working With Data . . . 20
4.5.2 Part 2: Checkouts, Branching, & Merging . . . 30
4.6 Hangar Under The Hood . . . 42
4.6.1 Things In Life Change, Your Data Shouldn't . . . 42
4.6.2 Data Is Large, We Don't Waste Space . . . 43
4.6.3 The Basics of Collaboration: Branching and Merging . . . 43
4.7 Python API . . . 47
4.7.1 Repository . . . 47
4.7.2 Write Enabled Checkout . . . 53
4.7.3 Read Only Checkout . . . 66
4.7.4 ML Framework Dataloaders . . . 73
4.8 Hangar CLI Documentation . . . 75
4.8.1 hangar . . . 75
4.9 Frequently Asked Questions . . . 80

4.9.1 How can I get an Invite to the Hangar User Group? . . . 80
4.9.2 Data Integrity . . . 80
4.9.3 How Can a Hangar Repository be Backed Up? . . . 82
4.9.4 On Determining `Arrayset` Schema Sizes . . . 82
4.10 Backend selection . . . 83
4.10.1 Identification . . . 83
4.10.2 Process & Guarantees . . . 83
4.10.3 Backend Specifications . . . 84
4.11 Contributing . . . 88
4.11.1 Bug reports . . . 89
4.11.2 Documentation improvements . . . 89
4.11.3 Feature requests and feedback . . . 89
4.11.4 Development . . . 89
4.12 Contributor Code of Conduct . . . 90
4.12.1 Our Pledge . . . 90
4.12.2 Our Standards . . . 90
4.12.3 Our Responsibilities . . . 91
4.12.4 Scope . . . 91
4.12.5 Enforcement . . . 91
4.12.6 Attribution . . . 91
4.13 Authors . . . 91
4.14 Change Log . . . 91
4.14.1 **‘v0.2.0‘_** (2019-08-09) . . . 91
4.14.2 v0.1.1 (2019-05-24) . . . 92
4.14.3 v0.1.0 (2019-05-24) . . . 93
4.14.4 v0.0.0 (2019-04-15) . . . 93

5 Indices and tables **95**

Python Module Index **97**

Index **99**

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 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/

CHAPTER 2

Installation

Hangar is in early alpha development release!

```
pip install hangar
```

CHAPTER 3

Documentation

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

CHAPTER 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`

4.1 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 instal 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 .
```

4.4 Hangar Core Concepts

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 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!

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) Arraysets 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 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: **Arraysets**.

Abstraction 2: What Makes up a Arrayset?

A `Dataset` is made of one or more `Arraysets` (and optionally some `Metadata`), with each item placed in some `Arrayset` 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 `Arraysets`. To define a `Arrayset` in Hangar, we need only 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 "Arraysets" are both individually and collectively referred to as `Samples` in the Hangar vernacular. According to the specification above, all samples contained in a `Arrayset` 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 `arrayset` specification (more on this later).

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

Effective use of Hangar relies on having an understanding of what exactly a `"Sample"` is in a particular `Arrayset`. 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 `arrayset`, this would be a single digit image (a 28x28 array); for a medical `arrayset` 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
Arraysets, 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
                         |
      ---------------------------------------
      |            |            |             |
  Arrayset 1   Arrayset 2   Arrayset 3    Arrayset 4
      |            |            |             |
-------------------------------------------------------
    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 Arraysets. 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 arraysets, 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 arrayset 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 arraysets containing information they care about, then they will.* Open source software development benefits everyone, we believe open source arrayset curation can do the same.

How To Overcome The "Size" Problem

Even if the greatest tool imaginable existed to version, branch, and merge arraysets, 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 arraysets 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, arraysets, 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 guarrentees 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 is the "locating information" 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 arrayset 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 arrayset, sample, or some metadata which did not previously exist in the relevant branch history.

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

Mutation An operation which sets: data to a sample, the value of some metadata key, or a arrayset schema, to a different value than what it had previously been created with (Note: a arrayset schema mutation is observed when a arrayset is removed, and a new arrayset 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 / arraysets / metadata) with non-compatible values. For samples & metadata, the hash of the data is compared, for arraysets, 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, arraysets, and metadata identically.
3) **Mutation** in `Dev Commit / Branch` **& Removal** in `Master Commit / Branch`. Applies for samples, arraysets, and metadata identically.
4) **Mutations** on keys both branches to non-compatible values. For samples & metadata, the hash of the data is compared, for arraysets, 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 Hangar Tutorial

4.5.1 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 `Repository` object.

Weather 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 `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-
↪iniitlized 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')
```

[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 it's reps, and the most relevant information will be sent to your screen!

[42]:
```
co = repo.checkout(write=True)
co
```

[42]:
```
Hangar WriterCheckout
    Writer       : True
    Base Branch  : master
    Num Arraysets : 2
    Num Metadata : 0
```

A checkout allows access to `arraysets` and `metadata`

The `arrayset` and `metadata` attributes of a checkout provide the interface to working with all of the data on disk!

[43]:
```
co.arraysets
```

[43]:
```
Hangar Arraysets
    Writeable: True
    Arrayset Names / Partial Remote References:
      - train_images / False
      - train_labels / False
```

[5]:
```
co.metadata
```

[5]:
```
Hangar Metadata
    Writeable: True
    Number of Keys: 0
```

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

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

[9]:
```
# 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 arrayset must be initialized.

A Arrayset is a named grouping of data samples where each sample shares a number of similar attributes and array properties. See the docstrings in `co.arraysets.init_arrayset`:

```
Initializes a arrayset in the repository.

Arraysets are groups of related data pieces (samples). All samples within
a arrayset 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 samples written to the arrayset must have
the same size that was initially specified upon arrayset initialization.
Variable size arraysets on the other hand, can write samples with
dimensions of any size less than a maximum which is required to be set
upon arrayset creation.

Parameters
----------
name : str
    The name assigned to this arrayset.
shape : Union[int, Tuple[int]]
    The shape of the data samples which will be written in this arrayset.
    This argument and the `dtype` argument are required if a `prototype`
    is not provided, defaults to None.
dtype : np.dtype
    The datatype of this arrayset. This argument and the `shape` argument
```

(continues on next page)

(continued from previous page)

```
    are required if a `prototype` is not provided., defaults to None.
prototype : np.ndarray
    A sample array of correct datatype and shape which will be used to
    initialize the arrayset storage mechanisms. If this is provided, the
    `shape` and `dtype` arguments must not be set, defaults to None.
named_samples : bool, optional
    If the samples in the arrayset have names associated with them. If set,
    all samples must be provided names, if not, no name will be assigned.
    defaults to True, which means all samples should have names.
variable_shape : bool, optional
    If this is a variable sized arrayset. If true, a the maximum shape is
    set from the provided `shape` or `prototype` argument. Any sample
    added to the arrayset can then have dimension sizes <= to this
    initial specification (so long as they have the same rank as what
    was specified) defaults to False.
backend : DEVELOPER USE ONLY. str, optional, kwarg only
    Backend which should be used to write the arrayset files on disk.

Returns
-------
:class:`ArraysetDataWriter`
    instance object of the initialized arrayset.

Raises
------
ValueError
    If provided name contains any non ascii, non alpha-numeric characters.
ValueError
    If required `shape` and `dtype` arguments are not provided in absence of
    `prototype` argument.
ValueError
    If `prototype` argument is not a C contiguous ndarray.
ValueError
    If rank of maximum tensor shape > 31.
ValueError
    If zero sized dimension in `shape` argument
ValueError
    If the specified backend is not valid.
```

[8]:
```
co.arraysets.init_arrayset(name='mnist_training_images', prototype=trimgs[0])
```

[8]:
```
Hangar ArraysetDataWriter
    Arrayset Name             : mnist_training_images
    Schema Hash               : 976ba57033bb
    Variable Shape            : False
    (max) Shape               : (784,)
    Datatype                  : <class 'numpy.uint8'>
    Named Samples             : True
    Access Mode               : a
    Number of Samples         : 0
    Partial Remote Data Refs : False
```

[9]:
```
co.arraysets['mnist_training_images']
```

[9]:
```
Hangar ArraysetDataWriter
    Arrayset Name             : mnist_training_images
```

(continues on next page)

(continued from previous page)

```
    Schema Hash               : 976ba57033bb
    Variable Shape            : False
    (max) Shape               : (784,)
    Datatype                  : <class 'numpy.uint8'>
    Named Samples             : True
    Access Mode               : a
    Number of Samples         : 0
    Partial Remote Data Refs : False
```

Interaction

When a arrayset is initialized, a arrayset 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 object. 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).

[10]:
```
co.arraysets['mnist_training_images']
```

[10]:
```
Hangar ArraysetDataWriter
    Arrayset Name             : mnist_training_images
    Schema Hash               : 976ba57033bb
    Variable Shape            : False
    (max) Shape               : (784,)
    Datatype                  : <class 'numpy.uint8'>
    Named Samples             : True
    Access Mode               : a
    Number of Samples         : 0
    Partial Remote Data Refs : False
```

[11]:
```
train_aset = co.arraysets['mnist_training_images']
```

the full dictionary style mapping interface is implemented

Adding Data

To add data to a named arrayset, we can use dict-style setting, or the `.add` method.

[12]:
```
train_aset['0'] = trimgs[0]
train_aset.add(data=trimgs[1], name='1')

train_aset['51'] = trimgs[51]
```

How many samples are in the arrayset?

[13]:
```
len(train_aset)
```

[13]:
```
3
```

Containment Testing

[14]:
```
'hi' in train_aset
```

[14]:
```
False
```

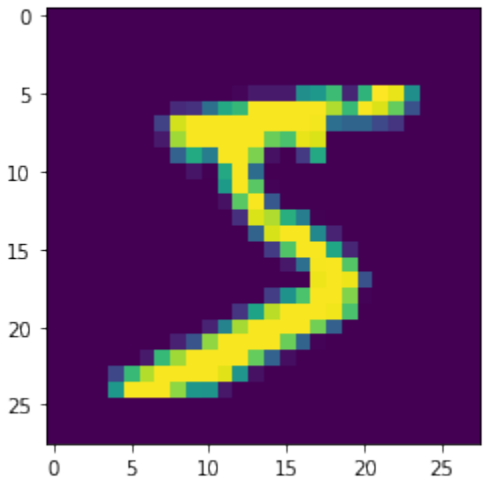
[15]:
```
'0' in train_aset
```

[15]:
```
True
```

Dictionary Style Retrieval for known keys

[16]:
```
out = train_aset['0']
plt.imshow(out.reshape(28, 28))
```

[16]:
```
<matplotlib.image.AxesImage at 0x11c16b9b0>
```



[17]:
```
train_aset
```

[17]:
```
Hangar ArraysetDataWriter
    Arrayset Name             : mnist_training_images
    Schema Hash               : 976ba57033bb
    Variable Shape            : False
    (max) Shape               : (784,)
    Datatype                  : <class 'numpy.uint8'>
    Named Samples             : True
    Access Mode               : a
    Number of Samples         : 3
    Partial Remote Data Refs : False
```

Dict style iteration supported out of the box

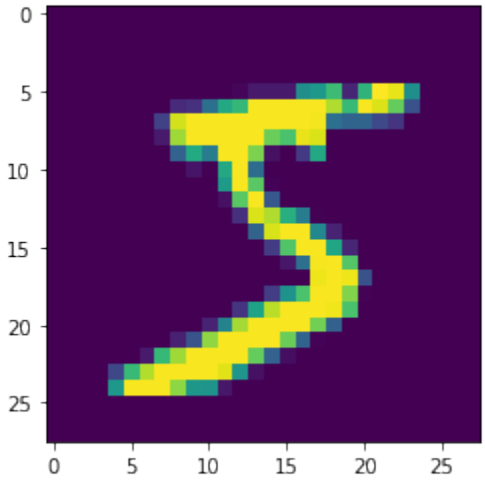
[18]:
```
for k in train_aset:
    # equivalent method: for k in train_aset.keys():
    print(k)
```

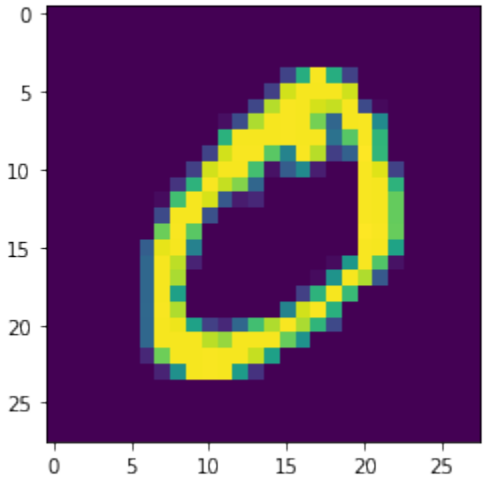
(continues on next page)

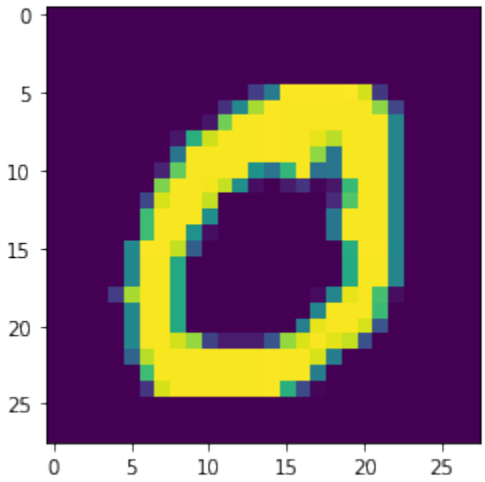
(continued from previous page)

```
for v in train_aset.values():
    plt.imshow(v.reshape(28, 28))
    plt.show()

myDict = {}
for k, v in train_aset.items():
    myDict[k] = v
```

```
0
1
51
```







Performance

Once you've completed an interactive exploration, be sure to use the context manager form of the `.add` 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 arraysets 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!

[10]:
```
import time

# ------------------ Context Manager Form ----------------------


aset_trimgs = co.arraysets.init_arrayset(name='train_images', prototype=sample_trimg)
aset_trlabels = co.arraysets.init_arrayset(name='train_labels', prototype=sample_
↪trlabel)

print(f'beginning non-context manager form')
start_time = time.time()
pbar = tqdm(total=trimgs.shape[0], leave=True)
for idx, img in enumerate(trimgs):
    if (idx % 500 == 0):
        pbar.update(500)

    aset_trimgs.add(data=img, name=str(idx))
    aset_trlabels.add(data=np.array([trlabels[idx]]), name=str(idx))

pbar.close()
print(f'Finished non-context manager form in: {time.time() - start_time} seconds')

co.reset_staging_area()
co.close()
```

(continues on next page)

(continued from previous page)

```
# ----------------- Non-Context Manager Form ----------------------

co = repo.checkout(write=True)

aset_trimgs = co.arraysets.init_arrayset(name='train_images', prototype=sample_trimg)
aset_trlabels = co.arraysets.init_arrayset(name='train_labels', prototype=sample_
↪trlabel)

print(f'beginning context manager form')
start_time = time.time()
pbar = tqdm(total=trimgs.shape[0], leave=True)
with aset_trimgs, aset_trlabels:
    for idx, img in enumerate(trimgs):
        if (idx % 500 == 0):
            pbar.update(500)

        aset_trimgs.add(data=img, name=str(idx))
        aset_trlabels.add(data=np.array([trlabels[idx]]), name=str(idx))

        pbar.close()
print(f'Finished context manager form in: {time.time() - start_time} seconds')
```

```
  0%|          | 0/50000 [00:00<?, ?it/s]
beginning non-context manager form
100%|| 50000/50000 [01:29<00:00, 570.33it/s]
Finished non-context manager form in: 90.59357118606567 seconds
  1%|          | 500/50000 [00:00<00:05, 9598.43it/s]
beginning context manager form

Finished context manager form in: 20.370257139205933 seconds
```

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 wan't to commit, just simply call the `commit()` method (and pass in a message)!

```
[21]: co.commit('hello world, this is my first hangar commit')
```

```
[21]: 'd2a9e7559252fba729694dd31f3474710b9153e9'
```

The returned value (`'d2a9e7559252fba729694dd31f3474710b9153e9'`) 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! (Branching to be covered in the next round of tutorials)

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 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()`!

[34]:
```
# Just call...

co.close()
```

But if you did forget, and you recieve a `PermissionError` next time you open a checkout

```
PermissionError: Cannot aquire 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.
```

This is a dangerous operation, and is 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).

[35]:
```
repo.force_release_writer_lock()
```

[35]:
```
True
```

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!

[36]:
```
repo.summary()
```

```
Summary of Contents Contained in Data Repository

==================
| Repository Info
|-----------
|  Base Directory: /Users/rick/projects/tensorwerk/hangar/dev/mnist
|  Disk Usage: 69.93 MB

===================
| Commit Details
-------------
|  Commit: 25a6ca6b84112ab209c1916c29ee1075fe8a6b52
|  Created: Mon Aug  5 23:44:26 2019
|  By: Rick Izzo
|  Email: rick@tensorwerk.com
|  Message: commit on testbranch

==================
| DataSets
|-----------
|  Number of Named Arraysets: 3
|
|  * Arrayset Name: mnist_training_images
```

(continues on next page)

(continued from previous page)

```
|    Num Arrays: 3
|    Details:
|    - schema_hash: 976ba57033bb
|    - schema_dtype: 2
|    - schema_is_var: False
|    - schema_max_shape: (784,)
|    - schema_is_named: True
|    - schema_default_backend: 00
|
|  * Arrayset Name: train_images
|    Num Arrays: 50000
|    Details:
|    - schema_hash: 976ba57033bb
|    - schema_dtype: 2
|    - schema_is_var: False
|    - schema_max_shape: (784,)
|    - schema_is_named: True
|    - schema_default_backend: 00
|
|  * Arrayset Name: train_labels
|    Num Arrays: 50000
|    Details:
|    - schema_hash: 631f0f57c469
|    - schema_dtype: 7
|    - schema_is_var: False
|    - schema_max_shape: (1,)
|    - schema_is_named: True
|    - schema_default_backend: 10

==================
| Metadata:
|-----------
|  Number of Keys: 1
```

[37]:
```
repo.log()
```

```
* 25a6ca6b84112ab209c1916c29ee1075fe8a6b52 (testbranch) : commit on testbranch
* d2a9e7559252fba729694dd31f3474710b9153e9 : hello world, this is my first hangar␣
↪commit
```

4.5.2 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 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='foo/pth')
```

[3]:
```
repo_pth = repo.init(user_name='Test User', user_email='test@foo.com')
```

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 potion of otherwise standard operations will generally flat out refuse to to execute (ie. read-only checkouts, log, push, etc.) until the first commit is made.

One of the only options available at this point in time is to create a write-enabled checkout on the `"master"` branch and 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 arraysets or metadata samples recorded in the checkout.

[7]:
```
print(f'number of metadata keys: {len(co.metadata)}')
print(f'number of arraysets: {len(co.arraysets)}')
```

```
number of metadata keys: 0
number of arraysets: 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 on historical record (commit) in the repo.

[8]:
```
dummy = np.arange(10, dtype=np.uint16)
aset = co.arraysets.init_arrayset(name='dummy_arrayset', prototype=dummy)
aset['0'] = dummy
initialCommitHash = co.commit('first commit with a single sample added to a dummy␣
↪arrayset')
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()
```

```
* 0fd892b7ce9d9d0150c68bb5483876d58c28cbf1 (master) : first commit with a single␣
↪sample added to a dummy arrayset
```

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 though a `checkout`. There are two types of checkouts (each of which have different uses and abilities):

Checking out a branch/commit for reading: 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: 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 it's base `staging area`
- On commit, the branch which was checked out has it's `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 it's `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 `Git`; Hangar branches also have the same lightweight and performant properties which make working with `Git` branches so appealing.

In hangar, branch must always have a `name` and a `base_commit`. However, If no `base_commit` is specified, the current writer branch `HEAD` `commit` is used as the `base_commit` hash for the branch automatically.

```
[10]: branch_1 = repo.create_branch(name='testbranch')
```

```
[11]: branch_1
```

```
[11]: 'testbranch'
```

viewing the log, we see that a new branch named: `testbranch` is pointing to our initial commit

[12]:
```
print(f'branch names: {repo.list_branches()} \n')
repo.log()
```

```
branch names: ['master', 'testbranch']

* 0fd892b7ce9d9d0150c68bb5483876d58c28cbf1 (master) (testbranch) : first commit with␣
↪a single sample added to a dummy arrayset
```

If instead, we do 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]:
```
'new'
```

[15]:
```
repo.log()
```

```
* 0fd892b7ce9d9d0150c68bb5483876d58c28cbf1 (master) (new) (testbranch) : first commit␣
↪with a single sample added to a dummy arrayset
```

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.arraysets
```

[17]:
```
Hangar Arraysets
    Writeable: True
    Arrayset Names / Partial Remote References:
      - dummy_arrayset / False
```

[18]:
```
co.arraysets['dummy_arrayset']
```

[18]:
```
Hangar ArraysetDataWriter
    Arrayset Name             : dummy_arrayset
    Schema Hash               : 43edf7aa314c
    Variable Shape            : False
    (max) Shape               : (10,)
    Datatype                  : <class 'numpy.uint16'>
    Named Samples             : True
    Access Mode               : a
    Number of Samples         : 1
    Partial Remote Data Refs : False
```

[19]:
```
co.arraysets['dummy_arrayset']['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.arraysets['dummy_arrayset']['1'] = arr
```

We can see that in this checkout, there are indeed, two samples in the `dummy_arrayset`

```
[21]: len(co.arraysets['dummy_arrayset'])
```

```
[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()
```

```
* 186f1ccae28ad8f58bcae95dd8c1115a3b0de9dd (new) : commit on `new` branch adding a␣
↪sample to dummy_arrayset
* 0fd892b7ce9d9d0150c68bb5483876d58c28cbf1 (master) (testbranch) : first commit with␣
↪a single sample added to a dummy arrayset
```

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!

[25]:
```
co.merge(message='message for commit (not used for FF merge)', dev_branch='new')
```

[25]:
```
'186f1ccae28ad8f58bcae95dd8c1115a3b0de9dd'
```

Let's check the log!

[26]:
```
repo.log()
```

```
* 186f1ccae28ad8f58bcae95dd8c1115a3b0de9dd (master) (new) : commit on `new` branch␣
↪adding a sample to dummy_arrayset
* 0fd892b7ce9d9d0150c68bb5483876d58c28cbf1 (testbranch) : first commit with a single␣
↪sample added to a dummy arrayset
```

[27]:
```
co.branch_name
```

[27]:
```
'master'
```

[28]:
```
co.commit_hash
```

[28]:
```
'186f1ccae28ad8f58bcae95dd8c1115a3b0de9dd'
```

[29]:
```
co.arraysets['dummy_arrayset']
```

[29]:
```
Hangar ArraysetDataWriter
    Arrayset Name             : dummy_arrayset
    Schema Hash               : 43edf7aa314c
    Variable Shape            : False
    (max) Shape               : (10,)
    Datatype                  : <class 'numpy.uint16'>
    Named Samples             : True
    Access Mode               : a
    Number of Samples         : 2
    Partial Remote Data Refs : False
```

As you can see, everything is as it should be!

[30]:
```
co.close()
```

Making a 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.arraysets
```

[32]:
```
Hangar Arraysets
    Writeable: True
    Arrayset Names / Partial Remote References:
      - dummy_arrayset / False
```

[33]:
```
co.arraysets['dummy_arrayset']
```

[33]:
```
Hangar ArraysetDataWriter
    Arrayset Name             : dummy_arrayset
    Schema Hash               : 43edf7aa314c
    Variable Shape            : False
    (max) Shape               : (10,)
    Datatype                  : <class 'numpy.uint16'>
    Named Samples             : True
    Access Mode               : a
    Number of Samples         : 1
    Partial Remote Data Refs : False
```

We will start by mutating sample `0` in `dummy_arrayset` to a different value

[34]:
```
dummy_aset = co.arraysets['dummy_arrayset']
```

[35]:
```
old_arr = dummy_aset['0']
new_arr = old_arr + 50
new_arr
```

[35]:
```
array([50, 51, 52, 53, 54, 55, 56, 57, 58, 59], dtype=uint16)
```

[36]:
```
dummy_aset['0'] = new_arr
```

let's make a commit here, then add some metadata and make a new commit (all on the `testbranch` branch)

[37]:
```
co.commit('mutated sample `0` of `dummy_arrayset` to new value')
```

[37]:
```
'2fe5c53a899ba6accbe8c19debd9a489e3baeaed'
```

[38]:
```
repo.log()
```

```
* 2fe5c53a899ba6accbe8c19debd9a489e3baeaed (testbranch) : mutated sample `0` of␣
↪`dummy_arrayset` to new value
* 0fd892b7ce9d9d0150c68bb5483876d58c28cbf1 : first commit with a single sample added␣
↪to a dummy arrayset
```

[39]:
```
co.metadata['hello'] = 'world'
```

[40]:
```
co.commit('added hellow world metadata')
```

[40]:
```
'836ba8ff1fe552fb65944e2340b2a2ef2b2b62d4'
```

[41]:
```
co.close()
```

Looking at our history how, we see that none of the original branches reference our first commit anymore

[42]:
```
repo.log()
```

```
* 836ba8ff1fe552fb65944e2340b2a2ef2b2b62d4 (testbranch) : added hellow world metadata
* 2fe5c53a899ba6accbe8c19debd9a489e3baeaed : mutated sample `0` of `dummy_arrayset` to new value
* 0fd892b7ce9d9d0150c68bb5483876d58c28cbf1 : first commit with a single sample added to a dummy arrayset
```

We can check the history of the "master" branch by specifying it as an argument to the log() method

[43]:
```
repo.log('master')
```

```
* 186f1ccae28ad8f58bcae95dd8c1115a3b0de9dd (master) (new) : commit on `new` branch adding a sample to dummy_arrayset
* 0fd892b7ce9d9d0150c68bb5483876d58c28cbf1 : first commit with a single sample added to a dummy arrayset
```

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 eachother (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

[44]:
```
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.

[45]:
```
co.merge(message='merge of testbranch into master', dev_branch='testbranch')
```

[45]:
```
'fd4a07ada0f138870924fc4ffee47839b77f1fbe'
```

If we now look at the log, we see that this has a much different look then before. The three way merge results in a history which references changes made in both diverged branches, and unifies them in a single commit

[46]:
```
repo.log()
```

```
*   fd4a07ada0f138870924fc4ffee47839b77f1fbe (master) : merge of testbranch into master
|\
| * 836ba8ff1fe552fb65944e2340b2a2ef2b2b62d4 (testbranch) : added hellow world metadata
| * 2fe5c53a899ba6accbe8c19debd9a489e3baeaed : mutated sample `0` of `dummy_arrayset` to new value
* | 186f1ccae28ad8f58bcae95dd8c1115a3b0de9dd (new) : commit on `new` branch adding a sample to dummy_arrayset
|/
* 0fd892b7ce9d9d0150c68bb5483876d58c28cbf1 : first commit with a single sample added to a dummy arrayset
```

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"`

```
[47]: co.arraysets
```

```
[47]: Hangar Arraysets
          Writeable: True
          Arrayset Names / Partial Remote References:
            - dummy_arrayset / False
```

```
[48]: co.arraysets['dummy_arrayset']
```

```
[48]: Hangar ArraysetDataWriter
          Arrayset Name             : dummy_arrayset
          Schema Hash               : 43edf7aa314c
          Variable Shape            : False
          (max) Shape               : (10,)
          Datatype                  : <class 'numpy.uint16'>
          Named Samples             : True
          Access Mode               : a
          Number of Samples         : 2
          Partial Remote Data Refs : False
```

```
[49]: co.arraysets['dummy_arrayset']['0']
```

```
[49]: array([50, 51, 52, 53, 54, 55, 56, 57, 58, 59], dtype=uint16)
```

```
[50]: co.arraysets['dummy_arrayset']['1']
```

```
[50]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10], dtype=uint16)
```

```
[51]: co.metadata
```

```
[51]: Hangar Metadata
          Writeable: True
          Number of Keys: 1
```

```
[52]: co.metadata['hello']
```

```
[52]: 'world'
```

The Merge was a success!

```
[53]: 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 / arraysets / metadata) with non-compatible values. For samples & metadata, the hash of the data is compared, for arraysets, 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, arraysets, and metadata identically.
3. **Mutation** in `Dev Commit/Branch` **& Removal** in `Master Commit / Branch`. Applies for samples, arraysets, and metadata identically.
4. **Mutations** on keys both branches to non-compatible values. For samples & metadata, the hash of the data is compared, for arraysets, 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!"`. If we then 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.

[54]:
```
co = repo.checkout(write=True, branch='new')
```

[55]:
```
co.metadata['hello'] = 'foo conflict... BOO!'
```

[56]:
```
co.commit ('commit on new branch to hello metadata key so we can demonstrate a␣
↪conflict')
```

[56]:
```
'0a0c4dbcfe63ce10fd2a87a98b785ce03099b09e'
```

[57]:
```
repo.log()
```

```
* 0a0c4dbcfe63ce10fd2a87a98b785ce03099b09e (new) : commit on new branch to hello␣
↪metadata key so we can demonstrate a conflict
* 186f1ccae28ad8f58bcae95dd8c1115a3b0de9dd : commit on `new` branch adding a sample␣
↪to dummy_arrayset
* 0fd892b7ce9d9d0150c68bb5483876d58c28cbf1 : first commit with a single sample added␣
↪to a dummy arrayset
```

When we attempt the merge, an exception is thrown telling us there is a conflict!

[58]:
```
co.merge(message='this merge should not happen', dev_branch='testbranch')
```

```
HANGAR VALUE ERROR:: Merge ABORTED with conflict: {'aset': ConflictRecords(t1=(),␣
↪t21=(), t22=(), t3=(), conflict=False), 'meta':␣
↪ConflictRecords(t1=(MetadataRecordKey(meta_name='hello'),), t21=(), t22=(), t3=(),␣
↪conflict=True), 'sample': {'dummy_arrayset': ConflictRecords(t1=(), t21=(), t22=(),␣
↪t3=(), conflict=False)}, 'conflict_found': True}
```

```
---------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-58-1a98dce1852b> 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)
    468             dev_branch_name=dev_branch,
    469             repo_path=self._repo_path,
-> 470             writer_uuid=self._writer_lock)
    471
    472         for asetHandle in self._arraysets.values():

~/projects/tensorwerk/hangar/hangar-py/src/hangar/merger.py in␣
↪select_merge_algorithm(message, branchenv, stageenv, refenv, stagehashenv,␣
↪master_branch_name, dev_branch_name, repo_path, writer_uuid)
    131
    132     except ValueError as e:
-> 133         raise e from None
    134
    135     finally:

~/projects/tensorwerk/hangar/hangar-py/src/hangar/merger.py in␣
↪select_merge_algorithm(message, branchenv, stageenv, refenv, stagehashenv,␣
↪master_branch_name, dev_branch_name, repo_path, writer_uuid)
    128                 refenv=refenv,
    129                 stagehashenv=stagehashenv,
-> 130                repo_path=repo_path)
    131
    132     except ValueError as e:

~/projects/tensorwerk/hangar/hangar-py/src/hangar/merger.py in␣
↪_three_way_merge(message, master_branch_name, masterHEAD, dev_branch_name, devHEAD,␣
↪ancestorHEAD, branchenv, stageenv, refenv, stagehashenv, repo_path)
    256     except ValueError as e:
    257         logger.error(e, exc_info=False)
-> 258         raise e from None
    259
    260     fmtCont = _merge_dict_to_lmdb_tuples(patchedRecs=mergeContents)

~/projects/tensorwerk/hangar/hangar-py/src/hangar/merger.py in␣
↪_three_way_merge(message, master_branch_name, masterHEAD, dev_branch_name, devHEAD,␣
↪ancestorHEAD, branchenv, stageenv, refenv, stagehashenv, repo_path)
    253
    254     try:
-> 255         mergeContents = _compute_merge_results(a_cont=aCont, m_cont=mCont,␣
↪d_cont=dCont)
    256     except ValueError as e:
    257         logger.error(e, exc_info=False)

~/projects/tensorwerk/hangar/hangar-py/src/hangar/merger.py in␣
↪_compute_merge_results(a_cont, m_cont, d_cont)
    350     if confs['conflict_found'] is True:
    351         msg = f'HANGAR VALUE ERROR:: Merge ABORTED with conflict: {confs}'
-> 352         raise ValueError(msg) from None
    353
    354     # merging: arrayset schemas
```

(continues on next page)

(continued from previous page)

```
ValueError: HANGAR VALUE ERROR:: Merge ABORTED with conflict: {'aset':␣
↪ConflictRecords(t1=(), t21=(), t22=(), t3=(), conflict=False), 'meta':␣
↪ConflictRecords(t1=(MetadataRecordKey(meta_name='hello'),), t21=(), t22=(), t3=(),␣
↪conflict=True), 'sample': {'dummy_arrayset': ConflictRecords(t1=(), t21=(), t22=(),␣
↪t3=(), conflict=False)}, 'conflict_found': True}
```

Checking for Conflicts

Alternatively, use the diff methods on a checkout to test for conflicts before attempting a merge.

[59]:
```
merge_results, conflicts_found = co.diff.branch('testbranch')
```

[60]:
```
conflicts_found
```

[60]:
```
{'aset': ConflictRecords(t1=(), t21=(), t22=(), t3=(), conflict=False),
 'meta': ConflictRecords(t1=(MetadataRecordKey(meta_name='hello'),), t21=(), t22=(),␣
↪t3=(), conflict=True),
 'sample': {'dummy_arrayset': ConflictRecords(t1=(), t21=(), t22=(), t3=(),␣
↪conflict=False)},
 'conflict_found': True}
```

[61]:
```
conflicts_found['meta']
```

[61]:
```
ConflictRecords(t1=(MetadataRecordKey(meta_name='hello'),), t21=(), t22=(), t3=(),␣
↪conflict=True)
```

The type codes for a `ConflictRecords` `namedtuple` such as the one we saw:

```
ConflictRecords(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

[62]:
```
del co.metadata['hello']
co.metadata['resolved'] = 'conflict by removing hello key'
co.commit('commit which removes conflicting metadata key')
```

[62]:
```
'9af80ed5df5d893b5e918f1a060cce4c46d9ddec'
```

[63]:
```
co.merge(message='this merge succeeds as it no longer has a conflict', dev_branch=
↪'testbranch')
```

[63]:
```
'b3b097d069f351e5b4688f1ebf30ae1a5aa94f4a'
```

We can verify that history looks as we would expect via the log!

[64]:
```
repo.log()
```

```
*   b3b097d069f351e5b4688f1ebf30ae1a5aa94f4a (new) : this merge succeeds as it no␣
↪longer has a conflict
|\
* | 9af80ed5df5d893b5e918f1a060cce4c46d9ddec : commit which removes conflicting␣
↪metadata key
* | 0a0c4dbcfe63ce10fd2a87a98b785ce03099b09e : commit on new branch to hello metadata␣
↪key so we can demonstrate a conflict
| * 836ba8ff1fe552fb65944e2340b2a2ef2b2b62d4 (testbranch) : added hellow world␣
↪metadata
| * 2fe5c53a899ba6accbe8c19debd9a489e3baeaed : mutated sample `0` of `dummy_arrayset`␣
↪to new value
* | 186f1ccae28ad8f58bcae95dd8c1115a3b0de9dd : commit on `new` branch adding a sample␣
↪to dummy_arrayset
|/
* 0fd892b7ce9d9d0150c68bb5483876d58c28cbf1 : first commit with a single sample added␣
↪to a dummy arrayset
```

4.6 Hangar Under The Hood

At it's core, hangar is a content addressable data store whose design requirements were inspired by the Git version control system.

4.6.1 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 arrayset 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.

4.6.2 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 *(arraysetname, 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 arrayset 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 arrayset, 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.

4.6.3 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'
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'
-------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
'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'
```

(continues on next page)

(continued from previous page)

```
# Commit Hash       |  Parent Commit
-------------------------------------
'479b4cfff6219e3d' ->  ''
```

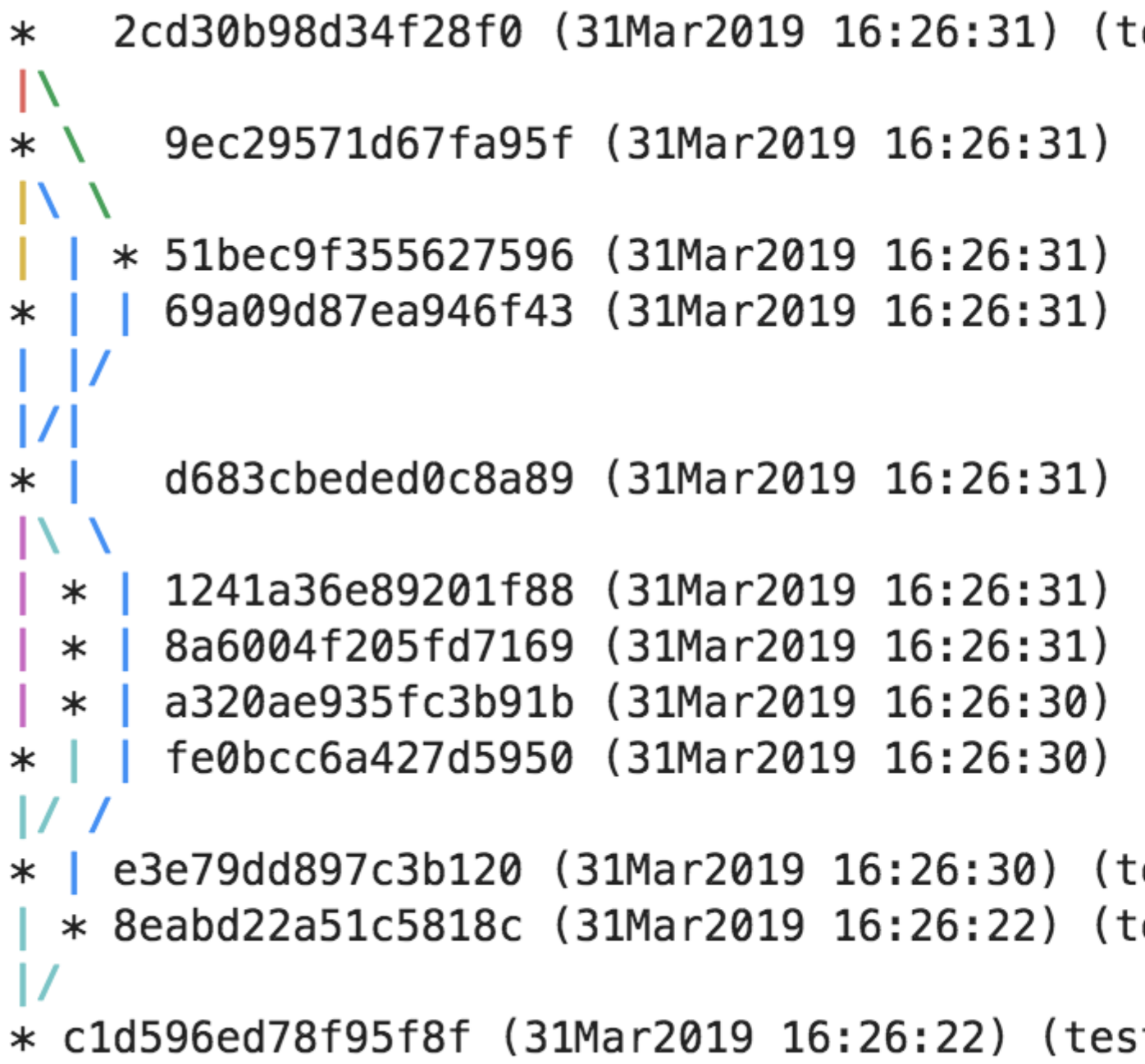
Branches can be created as cheaply a single line of text can be written, and they simply require a "root" commit hash (or a branch name, in which case the branch's current HEAD commit will be used as the root HEAD). Likewise a branch can be merged with just a single write operation (once the merge logic 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/')

... # make some commits

>>> repo.log()
```



4.7 Python API

This is the python API for the Hangar project.

4.7.1 Repository

class Repository(*path: os.PathLike*, *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.

Parameters

- **path** (*str*) – 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 = 'master'*, *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 'master'
- **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

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*) → str
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.

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 name of the branch which was created

Return type str

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*) → os.PathLike
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 os.PathLike

list_branches() → List[str]
list all branch names created in the repository.

Returns the branch names recorded in the repository

Return type list of 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

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

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

Return type os.PathLike

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*)
Not Implemented

summary(*, *branch: str = ''*, *commit: str = ''*, *return_contents: bool = False*) → Optional[dict]
Print a summary of the repository contents to the terminal

Note: Programatic access is provided by the return_contents argument.

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 = '')
- **return_contents** (*bool*) – If true, return a full log of what records are in the repository at the summary point. (Default value = False)

Returns contents of the entire repository (if *return_contents*=*True*)

Return type Optional[dict]

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

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

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 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*, ***, *arrayset_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
- **get_all** (*bool, optional*) – if data should be retrieved for all history accessible by the parents of this commit HEAD. by default False

Returns commit hashs 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.

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 = ''*) → bool
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 `ValueError` – 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

4.7.2 Write Enabled 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.

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.

arraysets
Provides access to arrayset interaction object.

See also:

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

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

Return type *Arraysets*

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.

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 string

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

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

Returns commit hash

Return type string

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*

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 (*master*)

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

Return type str

metadata
Provides access to metadata interaction object.

See also:

The class *hangar.metadata.MetadataWriter* contains all methods accessible by this property accessor

Returns weakref proxy to the metadata object which behaves exactly like a metadata class but which can be invalidated when the writer lock is released.

Return type *MetadataWriter*

reset_staging_area() → 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:`arrayset` 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.

Arraysets

class Arraysets

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

This object is the entry point to all tensor data stored in their individual arraysets. Each arrayset 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 tensors.

__contains__(*key: str*) → bool

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

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

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

Return type bool

__delitem__(*key: str*) → str

remove a arrayset and all data records if write-enabled process.

Parameters **key** (*str*) – Name of the arrayset 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 arrayset.

Return type str

Raises PermissionError – If this is a read-only checkout, no operation is permitted.

__getitem__(*key: str*) → Union[hangar.arrayset.ArraysetDataReader, hangar.arrayset.ArraysetDataWriter]

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

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

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

Return type *ArraysetDataReader* or *ArraysetDataWriter*

__setitem__(*key*, *value*)

Specifically prevent use dict style setting for arrayset objects.

Arraysets must be created using the factory function *init_arrayset()*.

Raises PermissionError – This operation is not allowed under any circumstance

contains_remote_references

Dict of bool indicating data reference locality in each arrayset.

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

Return type Mapping[str, bool]

get(*name: str*) → Union[hangar.arrayset.ArraysetDataReader, hangar.arrayset.ArraysetDataWriter]

Returns a arrayset access object.

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

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

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

Return type Union[*ArraysetDataReader*, *ArraysetDataWriter*]

Raises `KeyError` – If no arrayset with the given name exists in the checkout

init_arrayset(*name: str*, *shape: Union[int, Tuple[int]] = None*, *dtype: numpy.dtype = None*, *prototype: numpy.ndarray = None*, *named_samples: bool = True*, *variable_shape: bool = False*, *, *backend: str = None*) → hangar.arrayset.ArraysetDataWriter

Initializes a arrayset in the repository.

Arraysets are groups of related data pieces (samples). All samples within a arrayset 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 samples written to the arrayset must have the same size that was initially specified upon arrayset initialization. Variable size arraysets on the other hand, can write samples with dimensions of any size less than a maximum which is required to be set upon arrayset creation.

Parameters

- **name** (*str*) – The name assigned to this arrayset.
- **shape** (*Union[int, Tuple[int]]*) – The shape of the data samples which will be written in this arrayset. This argument and the *dtype* argument are required if a *prototype* is not provided, defaults to None.
- **dtype** (*np.dtype*) – The datatype of this arrayset. This argument and the *shape* argument are required if a *prototype* is not provided., defaults to None.
- **prototype** (*np.ndarray*) – A sample array of correct datatype and shape which will be used to initialize the arrayset storage mechanisms. If this is provided, the *shape* and *dtype* arguments must not be set, defaults to None.
- **named_samples** (*bool, optional*) – If the samples in the arrayset have names associated with them. If set, all samples must be provided names, if not, no name will be assigned. defaults to True, which means all samples should have names.
- **variable_shape** (*bool, optional*) – If this is a variable sized arrayset. If true, a the maximum shape is set from the provided *shape* or *prototype* argument. Any sample added to the arrayset can then have dimension sizes <= to this initial specification (so long as they have the same rank as what was specified) defaults to False.
- **backend** (*DEVELOPER USE ONLY. str, optional, kwarg only*) – Backend which should be used to write the arrayset files on disk.

Returns instance object of the initialized arrayset.

Return type *ArraysetDataWriter*

Raises

- `ValueError` – If provided name contains any non ascii, non alpha-numeric characters.
- `ValueError` – If required *shape* and *dtype* arguments are not provided in absence of *prototype* argument.
- `ValueError` – If *prototype* argument is not a C contiguous ndarray.
- `LookupError` – If a arrayset already exists with the provided name.
- `ValueError` – If rank of maximum tensor shape > 31.
- `ValueError` – If zero sized dimension in *shape* argument

- `ValueError` – If the specified backend is not valid.

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

items() → Iterable[Tuple[str, Union[hangar.arrayset.ArraysetDataReader, hangar.arrayset.ArraysetDataWriter]]]
generator providing access to arrayset_name, *Arraysets*

Yields *Iterable[Tuple[str, Union[ArraysetDataReader, ArraysetDataWriter]]]* – returns two tuple of all all arrayset names/object pairs in the checkout.

keys() → List[str]
list all arrayset keys (names) in the checkout

Returns list of arrayset names

Return type List[str]

multi_add(*mapping: Mapping[str, numpy.ndarray]*) → str
Add related samples to un-named arraysets with the same generated key.

If you have multiple arraysets in a checkout whose samples are related to each other in some manner, there are two ways of associating samples together:

1) using named arraysets and setting each tensor in each arrayset to the same sample "name" using un-named arraysets.
2) using this "add" method. which accepts a dictionary of "arrayset names" as keys, and "tensors" (ie. individual samples) as values.

When method (2) - this method - is used, the internally generated sample ids will be set to the same value for the samples in each arrayset. That way a user can iterate over the arrayset key's in one sample, and use those same keys to get the other related tensor samples in another arrayset.

Parameters **mapping** (*Mapping[str, np.ndarray]*) – Dict mapping (any number of) arrayset names to tensor data (samples) which to add. The arraysets must exist, and must be set to accept samples which are not named by the user

Returns generated id (key) which each sample is stored under in their corresponding arrayset. This is the same for all samples specified in the input dictionary.

Return type str

Raises `KeyError` – If no arrayset with the given name exists in the checkout

remote_sample_keys
Determine arraysets samples names which reference remote sources.

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

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

remove_aset(*aset_name: str*) → str
remove the arrayset and all data contained within it from the repository.

Parameters **aset_name** (*str*) – name of the arrayset to remove

Returns name of the removed arrayset

Return type str

Raises `KeyError` – If a arrayset does not exist with the provided name

values() → Iterable[Union[hangar.arrayset.ArraysetDataReader, hangar.arrayset.ArraysetDataWriter]]
yield all arrayset object instances in the checkout.

Yields *Iterable[Union[ArraysetDataReader, ArraysetDataWriter]]* – Generator of ArraysetData accessor objects (set to read or write mode as appropriate)

Arrayset Data

class ArraysetDataWriter
Class implementing methods to write data to a arrayset.

Writer specific methods are contained here, and while read functionality is shared with the methods common to *ArraysetDataReader*. Write-enabled checkouts are not thread/process safe for either `writes` OR `reads`, a restriction we impose for `write-enabled` checkouts in order to ensure data integrity above all else.

See also:

ArraysetDataReader

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

Parameters **key** (*Union[str, int]*) – name to check if it is a sample in the arrayset

Returns True if key exists, else False

Return type bool

__delitem__(*key: Union[str, int]*) → Union[str, int]
Remove a sample from the arrayset. Convenience method to *remove()*.

See also:

remove()

Parameters **key** (*Union[str, int]*) – Name of the sample to remove from the arrayset

Returns Name of the sample removed from the arrayset (assuming operation successful)

Return type Union[str, int]

__getitem__(*key: Union[str, int]*) → numpy.ndarray
Retrieve a sample with a given key. Convenience method for dict style access.

See also:

get()

Parameters **key** (*Union[str, int]*) – sample key to retrieve from the arrayset

Returns sample array data corresponding to the provided key

Return type np.ndarray

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

Returns number of samples the arrayset contains

Return type int

__setitem__(*key: Union[str, int], value: numpy.ndarray*) → Union[str, int]
Store a piece of data in a arrayset. Convenience method to *add()*.

See also:

add()

Parameters

- **key** (*Union[str, int]*) – name of the sample to add to the arrayset
- **value** (*np.array*) – tensor data to add as the sample

Returns sample name of the stored data (assuming operation was successful)

Return type Union[str, int]

add(*data: numpy.ndarray*, *name: Union[str, int] = None*, ***kwargs*) → Union[str, int]
Store a piece of data in a arrayset

Parameters

- **data** (*np.ndarray*) – data to store as a sample in the arrayset.
- **name** (*Union[str, int], optional*) – name to assign to the same (assuming the arrayset accepts named samples), If str, can only contain alpha-numeric ascii characters (in addition to '-', '.', '_'). Integer key must be >= 0. by default None

Returns sample name of the stored data (assuming the operation was successful)

Return type Union[str, int]

Raises

- ValueError – If no *name* arg was provided for arrayset requiring named samples.
- ValueError – If input data tensor rank exceeds specified rank of arrayset samples.
- ValueError – For variable shape arraysets, if a dimension size of the input data tensor exceeds specified max dimension size of the arrayset samples.
- ValueError – For fixed shape arraysets, if input data dimensions do not exactly match specified arrayset dimensions.
- ValueError – If type of *data* argument is not an instance of np.ndarray.
- ValueError – If *data* is not "C" contiguous array layout.
- ValueError – If the datatype of the input data does not match the specified data type of the arrayset

contains_remote_references
Bool indicating if all samples exist locally or if some reference remote sources.

dtype
Datatype of the arrayset schema. Read-only attribute.

get(*name: Union[str, int]*) → numpy.ndarray
Retrieve a sample in the arrayset with a specific name.

The method is thread/process safe IF used in a read only checkout. Use this if the calling application wants to manually manage multiprocess logic for data retrieval. Otherwise, see the *get_batch()* method to retrieve multiple data samples simultaneously. This method uses multiprocess pool of workers (managed by hangar) to drastically increase access speed and simplify application developer workflows.

Note: in most situations, we have observed little to no performance improvements when using multi-threading. However, access time can be nearly linearly decreased with the number of CPU cores / workers if multiprocessing is used.

Parameters **name** (*Union[str, int]*) – Name of the sample to retrieve data for.

Returns Tensor data stored in the arrayset archived with provided name(s).

Return type np.ndarray

Raises KeyError – if the arrayset does not contain data with the provided name

get_batch(*names: Iterable[Union[str, int]], *, n_cpus: int = None, start_method: str = 'spawn'*) → List[numpy.ndarray]

Retrieve a batch of sample data with the provided names.

This method is (technically) thread & process safe, though it should not be called in parallel via multi-thread/process application code; This method has been seen to drastically decrease retrieval time of sample batches (as compared to looping over single sample names sequentially). Internally it implements a multiprocess pool of workers (managed by hangar) to simplify application developer workflows.

Parameters

- **name** (*Iterable[Union[str, int]]*) – list/tuple of sample names to retrieve data for.
- **n_cpus** (*int, kwarg-only*) – if not None, uses num_cpus / 2 of the system for retrieval. Setting this value to 1 will not use a multiprocess pool to perform the work. Default is None
- **start_method** (*str, kwarg-only*) – One of 'spawn', 'fork', 'forkserver' specifying the process pool start method. Not all options are available on all platforms. see python multiprocess docs for details. Default is 'spawn'.

Returns

Tensor data stored in the arrayset archived with provided name(s).

If a single sample name is passed in as the, the corresponding np.array data will be returned.

If a list/tuple of sample names are pass in the names argument, a tuple of size len(names) will be returned where each element is an np.array containing data at the position it's name listed in the names parameter.

Return type List[np.ndarray]

Raises KeyError – if the arrayset does not contain data with the provided name

iswriteable

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

items() → Iterator[Tuple[Union[str, int], numpy.ndarray]]

generator yielding two-tuple of (name, tensor), for every sample in the arrayset.

For write enabled checkouts, is technically possible to iterate over the arrayset object while adding/deleting data, in order to avoid internal python runtime errors (dictionary changed size during iteration we have to make a copy of they key list before beginning the loop.) While not necessary for read checkouts, we perform the same operation for both read and write checkouts in order in order to avoid differences.

Yields *Iterator[Tuple[Union[str, int], np.ndarray]]* – sample name and stored value for every sample inside the arrayset

keys() → Iterator[Union[str, int]]
generator which yields the names of every sample in the arrayset

For write enabled checkouts, is technically possible to iterate over the arrayset object while adding/deleting data, in order to avoid internal python runtime errors (`dictionary changed size during iteration` we have to make a copy of they key list before beginning the loop.) While not necessary for read checkouts, we perform the same operation for both read and write checkouts in order in order to avoid differences.

Yields *Iterator[Union[str, int]]* – keys of one sample at a time inside the arrayset

name
Name of the arrayset. Read-Only attribute.

named_samples
Bool indicating if samples are named. Read-only attribute.

remote_reference_sample_keys
Returns sample names whose data is stored in a remote server reference.

Returns list of sample keys in the arrayset.

Return type List[str]

remove(*name: Union[str, int]*) → Union[str, int]
Remove a sample with the provided name from the arrayset.

Note: This operation will NEVER actually remove any data from disk. If you commit a tensor at any point in time, **it will always remain accessible by checking out a previous commit** when the tensor was present. This is just a way to tell Hangar that you don't want some piece of data to clutter up the current version of the repository.

Warning: Though this may change in a future release, in the current version of Hangar, we cannot recover references to data which was added to the staging area, written to disk, but then removed **before** a commit operation was run. This would be a similar sequence of events as: checking out a *git* branch, changing a bunch of text in the file, and immediately performing a hard reset. If it was never committed, git doesn't know about it, and (at the moment) neither does Hangar.

Parameters **name** (*Union[str, int]*) – name of the sample to remove.

Returns If the operation was successful, name of the data sample deleted.

Return type Union[str, int]

Raises `KeyError` – If a sample with the provided name does not exist in the arrayset.

shape
Shape (or *max_shape*) of the arrayset sample tensors. Read-only attribute.

values() → Iterator[numpy.ndarray]
generator which yields the tensor data for every sample in the arrayset

For write enabled checkouts, is technically possible to iterate over the arrayset object while adding/deleting data, in order to avoid internal python runtime errors (`dictionary changed size during iteration` we have to make a copy of they key list before beginning the loop.) While not necessary

for read checkouts, we perform the same operation for both read and write checkouts in order in order to avoid differences.

Yields *Iterator[np.ndarray]* – values of one sample at a time inside the arrayset

variable_shape
Bool indicating if arrayset schema is variable sized. Read-only attribute.

Metadata

class MetadataWriter
Class implementing write access to repository metadata.

Similar to the *ArraysetDataWriter*, this class inherits the functionality of the *MetadataReader* for reading. The only difference is that the reader will be initialized with data records pointing to the staging area, and not a commit which is checked out.

Note: Write-enabled metadata objects are not thread or process safe. Read-only checkouts can use multithreading safety to retrieve data via the standard *MetadataReader.get()* calls

---

See also:

MetadataReader for the intended use of this functionality.

__contains__(*key: Union[str, int]*) → bool
Determine if a key with the provided name is in the metadata

Parameters **key** (*Union[str, int]*) – key to check for containment testing

Returns True if key exists, False otherwise

Return type bool

__delitem__(*key: Union[str, int]*) → Union[str, int]
Remove a key/value pair from metadata. Convenience method to *remove()*.

See also:

remove() for the function this calls into.

Parameters **key** (*Union[str, int]*) – Name of the metadata piece to remove.

Returns Metadata key removed from the checkout (assuming operation successful)

Return type Union[str, int]

__getitem__(*key: Union[str, int]*) → str
Retrieve a metadata sample with a key. Convenience method for dict style access.

See also:

get()

Parameters **key** (*Union[str, int]*) – metadata key to retrieve from the checkout

Returns value of the metadata key/value pair stored in the checkout.

Return type string

__len__() → int
Determine how many metadata key/value pairs are in the checkout

Returns number of metadata key/value pairs.

Return type int

__setitem__(*key: Union[str, int], value: str*) → Union[str, int]
Store a key/value pair as metadata. Convenience method to *add()*.

See also:

add()

Parameters

- **key** (*Union[str, int]*) – name of the key to add as metadata
- **value** (*string*) – value to add as metadata

Returns key of the stored metadata sample (assuming operation was successful)

Return type Union[str, int]

add(*key: Union[str, int], value: str*) → Union[str, int]
Add a piece of metadata to the staging area of the next commit.

Parameters

- **key** (*Union[str, int]*) – Name of the metadata piece, alphanumeric ascii characters only
- **value** (*string*) – Metadata value to store in the repository, any length of valid ascii characters.

Returns The name of the metadata key written to the database if the operation succeeded.

Return type Union[str, int]

Raises

- `ValueError` – If the *key* contains any whitespace or non alpha-numeric characters.
- `ValueError` – If the *value* contains any non ascii characters.

get(*key: Union[str, int]*) → str
retrieve a piece of metadata from the checkout.

Parameters **key** (*Union[str, int]*) – The name of the metadata piece to retrieve.

Returns The stored metadata value associated with the key.

Return type str

Raises

- `ValueError` – If the *key* is not str type or contains whitespace or non alpha-numeric characters.
- `KeyError` – If no metadata exists in the checkout with the provided key.

iswriteable
Read-only attribute indicating if this metadata object is write-enabled.

Returns True if write-enabled checkout, Otherwise False.

Return type bool

items() → Iterator[Tuple[Union[str, int], str]]
generator yielding key/value for all metadata recorded in checkout.

For write enabled checkouts, is technically possible to iterate over the metadata object while adding/deleting data, in order to avoid internal python runtime errors (`dictionary changed size during iteration` we have to make a copy of they key list before beginning the loop.) While not necessary for read checkouts, we perform the same operation for both read and write checkouts in order in order to avoid differences.

Yields *Iterator[Tuple[Union[str, int], np.ndarray]]* – metadata key and stored value for every piece in the checkout.

keys() → Iterator[Union[str, int]]
generator which yields the names of every metadata piece in the checkout.

For write enabled checkouts, is technically possible to iterate over the metadata object while adding/deleting data, in order to avoid internal python runtime errors (`dictionary changed size during iteration` we have to make a copy of they key list before beginning the loop.) While not necessary for read checkouts, we perform the same operation for both read and write checkouts in order in order to avoid differences.

Yields *Iterator[Union[str, int]]* – keys of one metadata sample at a time

remove(*key: Union[str, int]*) → Union[str, int]
Remove a piece of metadata from the staging area of the next commit.

Parameters **key** (*Union[str, int]*) – Metadata name to remove.

Returns Name of the metadata key/value pair removed, if the operation was successful.

Return type Union[str, int]

Raises

- `ValueError` – If the key provided is not string type and containing only ascii-alphanumeric characters.
- `KeyError` – If the checkout does not contain metadata with the provided key.

values() → Iterator[str]
generator yielding all metadata values in the checkout

For write enabled checkouts, is technically possible to iterate over the metadata object while adding/deleting data, in order to avoid internal python runtime errors (`dictionary changed size during iteration` we have to make a copy of they key list before beginning the loop.) While not necessary for read checkouts, we perform the same operation for both read and write checkouts in order in order to avoid differences.

Yields *Iterator[str]* – values of one metadata piece at a time

Differ

class WriterUserDiff

branch(*dev_branch_name: str*) → tuple
Compute changes and conflicts for diff between HEAD and a branch name.

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

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

Return type tuple

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

commit (*dev_commit_hash: str*) → tuple
Compute changes and conflicts for diff between HEAD and a commit hash.

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

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

Return type tuple

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

staged()
Return the diff of the staging area contents to the staging base HEAD

Returns contains all *addition*, *mutation*, *removals* and *unchanged* arraysets schema, samples, and metadata references in the current staging area.

Return type dict

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 arrayset or metadata record has changed however, the status is "DIRTY".

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

Return type str

4.7.3 Read Only Checkout

class ReaderCheckout
Checkout the repository as it exists at a particular branch.

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.

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.

arraysets
Provides access to arrayset interaction object.

See also:

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

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

Return type *Arraysets*

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.

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

Returns commit hash of the checkout

Return type string

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*

metadata
Provides access to metadata interaction object.

See also:

The class *hangar.metadata.MetadataReader* contains all methods accessible by this property accessor

Returns weakref proxy to the metadata object which behaves exactly like a metadata class but which can be invalidated when the writer lock is released.

Return type *MetadataReader*

Arraysets

class Arraysets
Common access patterns and initialization/removal of arraysets in a checkout.

This object is the entry point to all tensor data stored in their individual arraysets. Each arrayset 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 tensors.

__contains__(*key: str*) → bool
Determine if a arrayset with a particular name is stored in the checkout

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

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

Return type bool

__getitem__(*key: str*) → Union[hangar.arrayset.ArraysetDataReader, hangar.arrayset.ArraysetDataWriter]
Dict style access to return the arrayset object with specified key/name.

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

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

Return type *ArraysetDataReader* or *ArraysetDataWriter*

get(*name: str*) → Union[hangar.arrayset.ArraysetDataReader, hangar.arrayset.ArraysetDataWriter]
Returns a arrayset access object.

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

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

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

Return type Union[*ArraysetDataReader*, *ArraysetDataWriter*]

Raises KeyError – If no arrayset with the given name exists in the checkout

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

items() → Iterable[Tuple[str, Union[hangar.arrayset.ArraysetDataReader, hangar.arrayset.ArraysetDataWriter]]]
generator providing access to arrayset_name, *Arraysets*

Yields *Iterable[Tuple[str, Union[ArraysetDataReader, ArraysetDataWriter]]]* – returns two tuple of all all arrayset names/object pairs in the checkout.

keys() → List[str]
list all arrayset keys (names) in the checkout

Returns list of arrayset names

Return type List[str]

values() → Iterable[Union[hangar.arrayset.ArraysetDataReader, hangar.arrayset.ArraysetDataWriter]]
yield all arrayset object instances in the checkout.

Yields *Iterable[Union[ArraysetDataReader, ArraysetDataWriter]]* – Generator of ArraysetData accessor objects (set to read or write mode as appropriate)

Arrayset Data

class ArraysetDataReader
Class implementing get access to data in a arrayset.

The methods implemented here are common to the *ArraysetDataWriter* accessor class as well as to this "read-only" method. Though minimal, the behavior of read and write checkouts is slightly unique, with the main difference being that "read-only" checkouts implement both thread and process safe access methods. This is not possible for "write-enabled" checkouts, and attempts at multiprocess/threaded writes will generally fail with cryptic error messages.

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

Parameters **key** (*Union[str, int]*) – name to check if it is a sample in the arrayset

Returns True if key exists, else False

Return type bool

__getitem__(*key: Union[str, int]*) → numpy.ndarray
Retrieve a sample with a given key. Convenience method for dict style access.

See also:

get()

Parameters **key** (*Union[str, int]*) – sample key to retrieve from the arrayset

Returns sample array data corresponding to the provided key

Return type np.ndarray

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

Returns number of samples the arrayset contains

Return type int

contains_remote_references
Bool indicating if all samples exist locally or if some reference remote sources.

dtype
Datatype of the arrayset schema. Read-only attribute.

get(*name: Union[str, int]*) → numpy.ndarray
Retrieve a sample in the arrayset with a specific name.

The method is thread/process safe IF used in a read only checkout. Use this if the calling application wants to manually manage multiprocess logic for data retrieval. Otherwise, see the *get_batch()* method to retrieve multiple data samples simultaneously. This method uses multiprocess pool of workers (managed by hangar) to drastically increase access speed and simplify application developer workflows.

Note: in most situations, we have observed little to no performance improvements when using multithreading. However, access time can be nearly linearly decreased with the number of CPU cores / workers if multiprocessing is used.

Parameters **name** (*Union[str, int]*) – Name of the sample to retrieve data for.

Returns Tensor data stored in the arrayset archived with provided name(s).

Return type np.ndarray

Raises KeyError – if the arrayset does not contain data with the provided name

get_batch(*names: Iterable[Union[str, int]], *, n_cpus: int = None, start_method: str = 'spawn'*) → List[numpy.ndarray]
Retrieve a batch of sample data with the provided names.

This method is (technically) thread & process safe, though it should not be called in parallel via multithread/process application code; This method has been seen to drastically decrease retrieval time of sample batches (as compared to looping over single sample names sequentially). Internally it implements a multiprocess pool of workers (managed by hangar) to simplify application developer workflows.

Parameters

- **name** (*Iterable[Union[str, int]]*) – list/tuple of sample names to retrieve data for.
- **n_cpus** (*int, kwarg-only*) – if not None, uses num_cpus / 2 of the system for retrieval. Setting this value to `1` will not use a multiprocess pool to perform the work. Default is None
- **start_method** (*str, kwarg-only*) – One of 'spawn', 'fork', 'forkserver' specifying the process pool start method. Not all options are available on all platforms. see python multiprocess docs for details. Default is 'spawn'.

Returns

Tensor data stored in the arrayset archived with provided name(s).

If a single sample name is passed in as the, the corresponding np.array data will be returned.

If a list/tuple of sample names are pass in the `names` argument, a tuple of size `len(names)` will be returned where each element is an np.array containing data at the position it's name listed in the `names` parameter.

Return type List[np.ndarray]

Raises `KeyError` – if the arrayset does not contain data with the provided name

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

items() → Iterator[Tuple[Union[str, int], numpy.ndarray]]
generator yielding two-tuple of (name, tensor), for every sample in the arrayset.

For write enabled checkouts, is technically possible to iterate over the arrayset object while adding/deleting data, in order to avoid internal python runtime errors (`dictionary changed size during iteration` we have to make a copy of they key list before beginning the loop.) While not necessary for read checkouts, we perform the same operation for both read and write checkouts in order in order to avoid differences.

Yields *Iterator[Tuple[Union[str, int], np.ndarray]]* – sample name and stored value for every sample inside the arrayset

keys() → Iterator[Union[str, int]]
generator which yields the names of every sample in the arrayset

For write enabled checkouts, is technically possible to iterate over the arrayset object while adding/deleting data, in order to avoid internal python runtime errors (`dictionary changed size during iteration` we have to make a copy of they key list before beginning the loop.) While not necessary for read checkouts, we perform the same operation for both read and write checkouts in order in order to avoid differences.

Yields *Iterator[Union[str, int]]* – keys of one sample at a time inside the arrayset

name
Name of the arrayset. Read-Only attribute.

named_samples
Bool indicating if samples are named. Read-only attribute.

remote_reference_sample_keys
Returns sample names whose data is stored in a remote server reference.

Returns list of sample keys in the arrayset.

Return type List[str]

shape
Shape (or *max_shape*) of the arrayset sample tensors. Read-only attribute.

values() → Iterator[numpy.ndarray]
generator which yields the tensor data for every sample in the arrayset

For write enabled checkouts, is technically possible to iterate over the arrayset object while adding/deleting data, in order to avoid internal python runtime errors (`dictionary changed size during iteration` we have to make a copy of they key list before beginning the loop.) While not necessary for read checkouts, we perform the same operation for both read and write checkouts in order in order to avoid differences.

Yields *Iterator[np.ndarray]* – values of one sample at a time inside the arrayset

variable_shape
Bool indicating if arrayset schema is variable sized. Read-only attribute.

Metadata

class MetadataReader
Class implementing get access to the metadata in a repository.

Unlike the *ArraysetDataReader* and *ArraysetDataWriter*, the equivalent Metadata classes do not need a factory function or class to coordinate access through the checkout. This is primarily because the metadata is only stored at a single level, and because the long term storage is must simpler than for array data (just write to a lmdb database).

Note: It is important to realize that this is not intended to serve as a general store large amounts of textual data, and has no optimization to support such use cases at this time. This should only serve to attach helpful labels, or other quick information primarily intended for human book-keeping, to the main tensor data!

Note: Write-enabled metadata objects are not thread or process safe. Read-only checkouts can use multithreading safety to retrieve data via the standard *MetadataReader.get()* calls

---

__contains__(*key: Union[str, int]*) → bool
Determine if a key with the provided name is in the metadata

Parameters **key** (*Union[str, int]*) – key to check for containment testing

Returns True if key exists, False otherwise

Return type bool

__getitem__(*key: Union[str, int]*) → str
Retrieve a metadata sample with a key. Convenience method for dict style access.

See also:

get()

Parameters **key** (*Union[str, int]*) – metadata key to retrieve from the checkout

Returns value of the metadata key/value pair stored in the checkout.

Return type string

__len__() → int
Determine how many metadata key/value pairs are in the checkout

Returns number of metadata key/value pairs.

Return type int

get(*key: Union[str, int]*) → str
retrieve a piece of metadata from the checkout.

Parameters **key** (*Union[str, int]*) – The name of the metadata piece to retrieve.

Returns The stored metadata value associated with the key.

Return type str

Raises

- `ValueError` – If the *key* is not str type or contains whitespace or non alpha-numeric characters.
- `KeyError` – If no metadata exists in the checkout with the provided key.

iswriteable
Read-only attribute indicating if this metadata object is write-enabled.

Returns True if write-enabled checkout, Otherwise False.

Return type bool

items() → Iterator[Tuple[Union[str, int], str]]
generator yielding key/value for all metadata recorded in checkout.

For write enabled checkouts, is technically possible to iterate over the metadata object while adding/deleting data, in order to avoid internal python runtime errors (`dictionary changed size during iteration` we have to make a copy of they key list before beginning the loop.) While not necessary for read checkouts, we perform the same operation for both read and write checkouts in order in order to avoid differences.

Yields *Iterator[Tuple[Union[str, int], np.ndarray]]* – metadata key and stored value for every piece in the checkout.

keys() → Iterator[Union[str, int]]
generator which yields the names of every metadata piece in the checkout.

For write enabled checkouts, is technically possible to iterate over the metadata object while adding/deleting data, in order to avoid internal python runtime errors (`dictionary changed size during iteration` we have to make a copy of they key list before beginning the loop.) While not necessary for read checkouts, we perform the same operation for both read and write checkouts in order in order to avoid differences.

Yields *Iterator[Union[str, int]]* – keys of one metadata sample at a time

values() → Iterator[str]
generator yielding all metadata values in the checkout

For write enabled checkouts, is technically possible to iterate over the metadata object while adding/deleting data, in order to avoid internal python runtime errors (`dictionary changed size during iteration` we have to make a copy of they key list before beginning the loop.) While not necessary for read checkouts, we perform the same operation for both read and write checkouts in order in order to avoid differences.

Yields *Iterator[str]* – values of one metadata piece at a time

Differ

class ReaderUserDiff

branch (*dev_branch_name: str*) → tuple
Compute changes and conflicts for diff between HEAD and a branch name.

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

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

Return type tuple

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

commit (*dev_commit_hash: str*) → tuple
Compute changes and conflicts for diff between HEAD and a commit hash.

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

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

Return type tuple

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

4.7.4 ML Framework Dataloaders

Tensorflow

make_tf_dataset (*arraysets*, *keys: Sequence[str] = None*, *index_range: slice = None*, *shuffle: bool = True*)
Uses the hangar arraysets to make a tensorflow dataset. It uses *from_generator* function from *tensorflow.data.Dataset* with a generator function that wraps all the hangar arraysets. 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

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

- **index_range** (*slice*) – A python slice object which will be used to find the subset of arrayset. 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'arrayset.shuffle()' on the returned arrayset

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.arraysets['mnist_data']
>>> target = co.arraysets['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(*arraysets*, *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*.

Parameters

- **arraysets** (*ArraysetDataReader* or Sequence) – A arrayset object, a tuple of arrayset object or a list of arrayset objects.
- **keys** (*Sequence[str]*) – An iterable collection of sample names. If given only those samples will fetched from the arrayset
- **index_range** (*slice*) – A python slice object which will be used to find the subset of arrayset. Argument *keys* takes priority over *range* i.e. if both are given, keys will be used and *range* will be ignored
- **field_names** (*list or tuple of str*) – An array of field names used as the *field_names* for the returned namedtuple. If not given, arrayset 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('.')
```

(continues on next page)

(continued from previous page)

```
>>> co = repo.checkout()
>>> aset = co.arraysets['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`

4.8 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

-v, --version
display the Hangar version currently installed

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

list

list all branch names

Includes both remote branches as well as local branches.

```
hangar branch list [OPTIONS]
```

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

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, --aset <aset>
specify any number of aset 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

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]

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

4.9 Frequently Asked Questions

The following documentation are taken from questions and comments on the Hangar User Group Slack Channel and over various Github issues.

4.9.1 How can I get an Invite to the Hangar User Group?

Just click on This Signup Link to get started.

4.9.2 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
```

(continues on next page)

(continued from previous page)

```
- 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*

4.9.3 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)

4.9.4 On Determining `Arrayset` 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 arrayset. 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 arrayset

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 arraysets 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 arraysets 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

---

4.10 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.

4.10.1 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.

4.10.2 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.

4.10.3 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})`; ie `(500, 10)` or `(500, 4, 3)`. 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.

Record Format

Fields Recorded for Each Array

- Format Code
- File UID
- Dataset Number (`0:COLLECTION_COUNT` dataset selection)
- Collection Index (`0:COLLECTION_SIZE` dataset subarray selection)
- Subarray Shape

Separators used

- `SEP_KEY: ":"`
- `SEP_HSH: "$"`
- `SEP_LST: " "`
- `SEP_SLC: "*"`

Examples

1) Adding the first piece of data to a file:
   - Array shape (Subarray Shape): (10)
   - File UID: "2HvGf9"
   - Dataset Number: "0"
   - Collection Index: 0

   `Record Data => "00:2HvGf9$0 0*10"`

1) Adding to a piece of data to a the middle of a file:
   - Array shape (Subarray Shape): (20, 2, 3)
   - File UID: "WzUtdu"
   - Dataset Number: "3"
   - Collection Index: 199

   `Record Data => "00:WzUtdu$3 199*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.

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".

Record Format

Fields Recorded for Each Array

- Format Code
- File UID
- Alder32 Checksum
- Collection Index (0:COLLECTION_SIZE subarray selection)
- Subarray Shape

Separators used

- `SEP_KEY: ":"`
- `SEP_HSH: "$"`
- `SEP_LST: " "`
- `SEP_SLC: "*"`

Examples

1) Adding the first piece of data to a file:
    - Array shape (Subarray Shape): (10)
    - File UID: "NJUUUK"
    - Alder32 Checksum: 900338819
    - Collection Index: 2

    `Record Data => '10:NJUUUK$900338819$2*10'`

1) Adding to a piece of data to a the middle of a file:
    - Array shape (Subarray Shape): (20, 2, 3)
    - File UID: "Mk23nl"
    - Alder32 Checksum: 2546668575
    - Collection Index: 199

    `Record Data => "10:Mk23nl$2546668575$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 `alder32` 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.

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

4.11 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*.

4.11.1 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.

4.11.2 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.

4.11.3 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 :)

4.11.4 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
```

4.12 Contributor Code of Conduct

4.12.1 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.

4.12.2 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

[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 ...

- 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

4.12.3 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.

4.12.4 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.

4.12.5 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.

4.12.6 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

4.13 Authors

- Richard Izzo - rick@tensorwerk.com
- Luca Antiga - luca@tensorwerk.com
- Sherin Thomas - sherin@tensorwerk.com

4.14 Change Log

4.14.1 '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.

4.14.2 v0.1.1 (2019-05-24)

Bug Fixes

- Fixed typo in README which was uploaded to PyPi

4.14.3 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.

4.14.4 v0.0.0 (2019-04-15)

- First Public Release of Hangar!

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

h

hangar.backends.hdf5_00, 84
hangar.backends.numpy_10, 86
hangar.backends.remote_50, 87
hangar.backends.selection, 83
hangar.repository, 47

Index

Symbols

-email <email>
    hangar-clone command line option, 76
    hangar-init command line option, 77
-ip <ip>
    hangar-server command line option, 79
-name <name>
    hangar-clone command line option, 76
    hangar-init command line option, 77
-overwrite
    hangar-clone command line option, 76
    hangar-init command line option, 77
    hangar-server command line option, 79
-port <port>
    hangar-server command line option, 79
-timeout <timeout>
    hangar-server command line option, 79
-a, -all-history
    hangar-fetch-data command line option, 77
-d, -aset <aset>
    hangar-fetch-data command line option, 77
-n, -nbytes <nbytes>
    hangar-fetch-data command line option, 77
-v, -version
    hangar command line option, 75
__contains__() (*ArraysetDataReader method*), 68
__contains__() (*ArraysetDataWriter method*), 59
__contains__() (*Arraysets method*), 56, 67
__contains__() (*MetadataReader method*), 71
__contains__() (*MetadataWriter method*), 63
__delitem__() (*ArraysetDataWriter method*), 59
__delitem__() (*Arraysets method*), 56
__delitem__() (*MetadataWriter method*), 63
__getitem__() (*ArraysetDataReader method*), 69
__getitem__() (*ArraysetDataWriter method*), 59
__getitem__() (*Arraysets method*), 56, 67
__getitem__() (*MetadataReader method*), 71
__getitem__() (*MetadataWriter method*), 63
__len__() (*ArraysetDataReader method*), 69
__len__() (*ArraysetDataWriter method*), 59
__len__() (*MetadataReader method*), 71
__len__() (*MetadataWriter method*), 63
__setitem__() (*ArraysetDataWriter method*), 59
__setitem__() (*Arraysets method*), 56
__setitem__() (*MetadataWriter method*), 64

A

add() (*ArraysetDataWriter method*), 60
add() (*MetadataWriter method*), 64
add() (*Remotes method*), 51
ADDRESS
    hangar-remote-add command line option, 78
ArraysetDataReader (*class in hangar.arrayset*), 68
ArraysetDataWriter (*class in hangar.arrayset*), 59
Arraysets (*class in hangar.arrayset*), 56, 67
arraysets (*ReaderCheckout attribute*), 66
arraysets (*WriterCheckout attribute*), 54

B

BRANCH
    hangar-fetch command line option, 76
    hangar-push command line option, 78
branch() (*ReaderUserDiff method*), 73
branch() (*WriterUserDiff method*), 65
branch_name (*WriterCheckout attribute*), 54

C

checkout() (*Repository method*), 48
clone() (*Repository method*), 48
close() (*ReaderCheckout method*), 67

close() (*WriterCheckout method*), 54
commit() (*ReaderUserDiff method*), 73
commit() (*WriterCheckout method*), 54
commit() (*WriterUserDiff method*), 66
commit_hash (*ReaderCheckout attribute*), 67
commit_hash (*WriterCheckout attribute*), 54
contains_remote_references (*ArraysetDataReader attribute*), 69
contains_remote_references (*ArraysetDataWriter attribute*), 60
contains_remote_references (*Arraysets attribute*), 56
create_branch() (*Repository method*), 49

D

diff (*ReaderCheckout attribute*), 67
diff (*WriterCheckout attribute*), 54
dtype (*ArraysetDataReader attribute*), 69
dtype (*ArraysetDataWriter attribute*), 60

F

fetch() (*Remotes method*), 52
fetch_data() (*Remotes method*), 52
force_release_writer_lock() (*Repository method*), 49

G

get() (*ArraysetDataReader method*), 69
get() (*ArraysetDataWriter method*), 60
get() (*Arraysets method*), 56, 68
get() (*MetadataReader method*), 72
get() (*MetadataWriter method*), 64
get_batch() (*ArraysetDataReader method*), 69
get_batch() (*ArraysetDataWriter method*), 61

H

hangar command line option
    -v, -version, 75
hangar-branch-create command line option
    NAME, 75
    STARTPOINT, 75
hangar-clone command line option
    -email <email>, 76
    -name <name>, 76
    -overwrite, 76
    REMOTE, 76
hangar-fetch command line option
    BRANCH, 76
    REMOTE, 76
hangar-fetch-data command line option
    -a, -all-history, 77
    -d, -aset <aset>, 77
    -n, -nbytes <nbytes>, 77
    REMOTE, 77
    STARTPOINT, 77
hangar-init command line option
    -email <email>, 77
    -name <name>, 77
    -overwrite, 77
hangar-log command line option
    STARTPOINT, 78
hangar-push command line option
    BRANCH, 78
    REMOTE, 78
hangar-remote-add command line option
    ADDRESS, 78
    NAME, 78
hangar-remote-remove command line option
    NAME, 79
hangar-server command line option
    -ip <ip>, 79
    -overwrite, 79
    -port <port>, 79
    -timeout <timeout>, 79
hangar-summary command line option
    STARTPOINT, 80
hangar.backends.hdf5_00 (*module*), 84
hangar.backends.numpy_10 (*module*), 86
hangar.backends.remote_50 (*module*), 87
hangar.backends.selection (*module*), 83
hangar.repository (*module*), 47

I

init() (*Repository method*), 49
init_arrayset() (*Arraysets method*), 57
iswriteable (*ArraysetDataReader attribute*), 70
iswriteable (*ArraysetDataWriter attribute*), 61
iswriteable (*Arraysets attribute*), 58, 68
iswriteable (*MetadataReader attribute*), 72
iswriteable (*MetadataWriter attribute*), 64
items() (*ArraysetDataReader method*), 70
items() (*ArraysetDataWriter method*), 61
items() (*Arraysets method*), 58, 68
items() (*MetadataReader method*), 72
items() (*MetadataWriter method*), 64

K

keys() (*ArraysetDataReader method*), 70
keys() (*ArraysetDataWriter method*), 62
keys() (*Arraysets method*), 58, 68
keys() (*MetadataReader method*), 72
keys() (*MetadataWriter method*), 65

L

list_all() (*Remotes method*), 52

list_branches() (*Repository method*), 50
log() (*Repository method*), 50

M

make_tf_dataset() (*in module hangar*), 73
make_torch_dataset() (*in module hangar*), 74
merge() (*Repository method*), 50
merge() (*WriterCheckout method*), 55
metadata (*ReaderCheckout attribute*), 67
metadata (*WriterCheckout attribute*), 55
MetadataReader (*class in hangar.metadata*), 71
MetadataWriter (*class in hangar.metadata*), 63
multi_add() (*Arraysets method*), 58

N

NAME
    hangar-branch-create command line option, 75
    hangar-remote-add command line option, 78
    hangar-remote-remove command line option, 79
name (*ArraysetDataReader attribute*), 70
name (*ArraysetDataWriter attribute*), 62
named_samples (*ArraysetDataReader attribute*), 70
named_samples (*ArraysetDataWriter attribute*), 62

P

path (*Repository attribute*), 50
ping() (*Remotes method*), 52
push() (*Remotes method*), 53

R

ReaderCheckout (*class in hangar.checkout*), 66
ReaderUserDiff (*class in hangar.diff*), 73
REMOTE
    hangar-clone command line option, 76
    hangar-fetch command line option, 76
    hangar-fetch-data command line option, 77
    hangar-push command line option, 78
remote (*Repository attribute*), 50
remote_reference_sample_keys (*ArraysetDataReader attribute*), 70
remote_reference_sample_keys (*ArraysetDataWriter attribute*), 62
remote_sample_keys (*Arraysets attribute*), 58
Remotes (*class in hangar.repository*), 51
remove() (*ArraysetDataWriter method*), 62
remove() (*MetadataWriter method*), 65
remove() (*Remotes method*), 53
remove_aset() (*Arraysets method*), 58
remove_branch() (*Repository method*), 51
Repository (*class in hangar.repository*), 47
reset_staging_area() (*WriterCheckout method*), 55

S

shape (*ArraysetDataReader attribute*), 70
shape (*ArraysetDataWriter attribute*), 62
staged() (*WriterUserDiff method*), 66
STARTPOINT
    hangar-branch-create command line option, 75
    hangar-fetch-data command line option, 77
    hangar-log command line option, 78
    hangar-summary command line option, 80
status() (*WriterUserDiff method*), 66
summary() (*Repository method*), 51

V

values() (*ArraysetDataReader method*), 71
values() (*ArraysetDataWriter method*), 62
values() (*Arraysets method*), 58, 68
values() (*MetadataReader method*), 72
values() (*MetadataWriter method*), 65
variable_shape (*ArraysetDataReader attribute*), 71
variable_shape (*ArraysetDataWriter attribute*), 63
version (*Repository attribute*), 51

W

writer_lock_held (*Repository attribute*), 51
WriterCheckout (*class in hangar.checkout*), 53
WriterUserDiff (*class in hangar.diff*), 65