Contents

1 Introduction

`cl-naive-store` is a log structured document store. Documents are loaded in-memory to give facilitate fast querying. Depending on how you use the

store documents will be lazy loaded. indexed, and written completely in Common Lisp.

The "naive" comes from the fact that data is persisted as plists in files to make them human and machine readable, also there is no rocket science code.

The store was designed to be customisable, just about anything can be customised, have a look at the implementation API to get an idea of what is possible.

1.1 Layered Design

`cl-naive-store` can do a lot but you as the user must decide how much of the store's functionality you want to use for your own project.

See Choosing Layers of Functionality for more information.

Functionality was broken down into these packages:

- `cl-naive-store.naive-core`
- `cl-naive-store.document-types`
- `cl-naive-store.definitions`
- `cl-naive-store.naive-documents`
- `cl-naive-store.naive-indexed`
- `cl-naive-store.tests`

The following .asd files can be used to load different functionality:

- `cl-naive-store.naive-core.asd` loads the most basic functionality for `cl-naive-store`. Use this if you don't any of the other extensions.
- `cl-naive-store.naive-merkle.asd` loads `naive-documents` and the *experimental* merkle functionality.
- `cl-naive-store.naive-indexed.asd` loads `naive-core` and index functionality.
- `cl-naive-store.document-types.asd` loads `naive-core` and document-type functionality.
- `cl-naive-store.document-defs.asd` loads `naive-core`, document-types and type definition functionality.

- `cl-naive-store.documents.asd` loads naive-core, naive-indexed, documents-types, definitions and document functionality.
- `cl-naive-store.asd` loads the whole shebang.

2 Releases

2.1 Version 2023.12.19

2.1.1 Multiverse

Added the concept of a multiverse to the store. The main reason was so you can set up more complex data schemas with references across stores.

2.1.2 Code Refactor

There use to be a lot of different methods to add, remove and persist stuff in naive store. This has been cleaned up to have just a couple of methods like add-multiverse-element. Have a look at the deprecation.lisp files for each layer for the details of what has been deprecated. We use cl-naive-deprecation to ease the transition, your code should run without you having to change anything immediately. Eventually you will have to update your code.

In general the code is cleaner and the interface simpler to learn.

2.1.3 Speed

Did work on cleaning up speed bottle necks.

2.1.4 Bugs

Many bugs where fixed during the conversion of the tests to cl-naive-tests.

2.1.5 Tests

cl-naive-store now uses cl-naive-tests and the tests where rewritten for better coverage and to be cleaner.

You will learn a lot from reading the test code.

3 Overview

3.1 Log Structured Database

At its core `cl-naive-store` is a log structured database.

A log-structured database is a type of database management system that organizes and stores its data in a sequential, append-only manner. Instead of overwriting data, it records all changes as a continuous stream of log entries. Each entry typically captures an operation, such as an insertion, update, or deletion, along with the associated data.

Advantages of a log-structured database include:

High Write Throughput: Since all changes are simply appended, the system can achieve high write speeds.

Simplified Crash Recovery: By replaying the log from a known good state, the database can recover its state after a crash. The replaying of the log however kicks you in the teeth when you get to tens of millions of records.

Consistent Snapshots: By marking points in the log, consistent snapshots of the database can be obtained without halting writes.

A log structured database can be slow to query if you query the files directly. To overcome that `cl-naive-store` was designed to be an in memory store. That means that when querying the data the file structure does not come into play.

`cl-naive-store` has several layers of functionality that extends the basic log store to a full document store. Each layer adds additional functionality at the cost of increased overhead and complexity.

The concept of a document is only loosely implemented and enforced in the base and indexed layers. It is when you use the full document layer that the store becomes a proper document store.

`cl-naive-store` does not enforce the use of data schemas for your data. Data schemas are optional and when used is only there to use as a guideline, no strict enforcement of data schemas are done. This in line with what is expected of a document store.

3.2 Choosing Layers of Functionality

Depending on the your use case you need to choose the appropriate layer(s) to use. It must be noted that you can use a mixture of layers depending on your requirements, this adds complexity to your data schemas and will take a lot of testing to get the right balance.

As a rule of thumb using the full document layer will serve most purposes well.

3.2.1 Base Layer (Simple Log Database)

The base layer `naive-core` implements the basics of a log store with in memory querying.

This layer has the smallest memory footprint per document and is the least prescriptive as far as the structure of the data you use.

When should you use just the base layer?

1. Small Database

   A couple hundred/thousand documents per collection. For example a database serving a simple website.

2. Large Database with no keys

   An example would be an event logger or such.

What you cannot do with the base layer is load millions of documents with keys. The simple reason for this is that the underlying structure (array) for the base layer does not support fast duplicate checking for keys. Duplicate checking is essential when loading (replaying) a log structured database from file.

This layer also does not have intelligence to deal with hierarchical data efficiently.

3.2.2 Indexed Layer

The indexed layer `naive-indexed` introduces document identity. It adds hash tables that support fast duplicate checking and lightning fast look-ups of individual documents or groups of documents.

So it is ideal for loading millions of records with keys.

This layer also does not have intelligence to deal with hierarchical data efficiently as it does not implement structured documents.

3.2.3 Document Layer

The document layer builds on all the other layers to implement a full document store. This layer wraps a document in a structure that allows it to recognise and deal with reference and associated documents, taking document identity to its conclusion. In this layer you can reference a document from one collection in another collection efficiently.

The document layer also introduces the concept of versions which plays to the strengths of a log structured database.

It is in this layer that you can start using data schemas (document types) for your collections.

3.3 Functionality by Layer

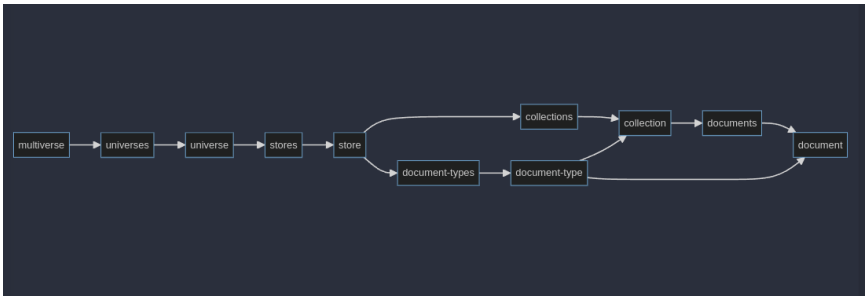
`cl-naive-store.*` packages are designed to be layered to achieve complex behaviour in incremental steps.

3.3.1 Functionality Matrix

Functionality	naive-core	naive-indexed	document-types	definitions	naiv
Load data into memory	T				
Delete Data	T				
Persist data to file	T				
Query Data	T				
Single Key Value	T				
Unique Object Identifier		T			
Multiple Key Values		T			
Key Value Lookups		T			
Index Data		T			
Index Lookups		T			
Handles Duplicates Properly		T			
Data Type aware Universe			T		
Data Schemas/Definitions				T	
Hierarchical Data Objects					T
Cross Collection Reference Objects					T
Object Version Tracking					T
Object Value Change Tracking					T

3.4 Structure of the store

The store has the following structure



3.4.1 Multiverse

A multiverse is the top structural container for data. A multiverse contains one or more universes. A multiverse could be viewed as a clustering of clusters of databases.

3.4.2 Universe

A universe contains one or more stores. A universe could be viewed as a cluster of databases.

3.4.3 Store

A store contains one or more collections. A store could be viewed as a database.

A store also contains one or more document-types.

3.4.4 Document Type

Document types are type schemas. A collection can be linked to a document-type. However not all document-types have a direct link to a collection. Some document types are indirectly linked because they are part of a document with a hierarchical structure.

3.4.5 Collection

When data is persisted the file folders/directories mirror the relationship above, which makes it possible to lazy load the data only when needed from disk. Querying an unloaded collection will cause the loading of a collection and in the case of naive-documents any referenced collections as well.

3.4.6 Documents

A document in `cl-naive-store` in simplest terms is a list of key value pairs, in other words a property list. This is also how a document is represented in the actual log files. Log files are read using cl read-line.

`naive-indexed` adds the concept of a UUID (aka hash key).

`naive-documents` adds additional meta data like the multiverse, universe, store, collection, changed data and old versions to the document.

3.5 In memory

Data is loaded into memory for querying and lookups, that makes them fast.

You can load the whole multiverse, universe, a store, a collection or a shard at a time.

3.6 Lazy Loading

You do not have to explicitly load data into memory upfront. You can leave it up to the store to only load data when needed. It means that you will only have the data that users requested up to that point in memory. Data in memory can easily be garbage collected if not in use any more. `cl-naive-store` does not do garbage collection for you that is left to the user.

3.7 Persistence

`cl-naive-store` relies on the fact that objects are translatable to key-value pairs and writes plists to a file per collection. Note of caution here if you go and store unprintable values (ie not readable) in the db you are going to be very disappointed when you try to load the db again!

3.8 Sharding

Sharding is the breaking down of files into smaller files, in the case of naive-store that means that instead of one file per collection there could be many.

Sharding is done based on the actual data in collections. The user specifies which elements of a document it wants to use for sharding on a collection. If none is specified no sharding is done.

You should not set up sharding to use data that can change for a document, it will cause problems. For instance if you use company name to affect sharding for data by company/client then you should not be able to change

the company name for the document. If you need to do it on rare occasions then you should delete (and write out a new log file with those documents stripped out) to keep your sanity.

3.9 Document Types

`cl-naive-store` is mostly/blissfully unaware of user defined document types and value types. `document-types` adds document-type and element classes, extending the store and collection classes to store document types.

Document types are ignored when doing persistence, and loading from disk, `document-types` just adds a place to store your document types and retrieve them at run time. Document types can be what ever you dream up!

If you want document type validation based on your document type definitions you need to implement it yourself, overriding add-object and persist-object should be enough.

3.10 Definitions

`cl-naive-store` stores definitions (plist trees) for multiverse, universe, store, collection and document-types. These can be user to load of a multiverse with minimal code instead of loading the store in code steps.

You can use `definitions` to define the full multiverse schema.

For an in depth discussion look at Definitions.

3.11 Naive Documents

Naive Documents uses `naive-core`, `naive-indexed`, `document-types` and `definitions` to create a more complex/fleshed out data store experience. Note that document types are still only used for their key and index definitions and no data type specific validation is done when loading or persisting data.

Other peculiarities of Naive Documents:

- Nothing stops you form adding "new" key values/types to your document at any time, since they are not validated against a document-type definition. A typical document database should be able to store different document types or at least document-types with varying data.
- A document has key-values that are used to check for equality when adding an object to a collection

- A document keeps a set of old and new values while you are updating values, this is cleared on persist.
- A document is expected to be hierarchical in nature, i.e. a document key-value pair can hold other documents (child documents). Child documents come in two basic flavours, documents that have no collection of their own (associated documents), and documents referenced from other collections (reference documents). When a top level document is persisted only "references" to the referenced child document are persisted (The child document itself is not updated!). Associated documents are persisted in full.

3.12 BLOBS

`cl-naive-store` knows how to deal with values that are blobs. Basically blobs are written to their own files and if file type is relevant the correct file type is used.

There are no tests for blobs yet so use at own risk!

4 Definitions

There are two main sources of definitions, hand crafted or persisted definitions.

Persisted definitions are found for individual elements along the directory structure of the peristed data and is relative to the multiverse/universe/store/collection structure. These definitions can be considered as partial definitions because they do not contain any information about their children. This is because they are created during add-multiverse-element and at that stage the element might not have any children yet. It also means that there is not one central complete definition that can show what the full store structure looks like.

A complete definition of all the multiverse elments is typically created by hand as a plist tree. You could of course create such a complete definition by walking the directory tree. A complete definition could be used for bootstrapping the database if needed. Bootstrapping the whole database is of course counter to the lazy loading principal that is supported by naive-store.

The definition manipulation functions supplied are simple in the sense that they will not ensure the integrity of the final definition. For example if you remove a document-type that document-type might still be used by a collection which will cause errors when loading from such a broken definition.

Definitions should always be a valid plist if you want to use the functions in naive-core to manipulate them.

This is the overview of what a full schema should look like.

```
(:multiverse
 [attributes]
 :universes ((:universe
              [attributes]
              :stores ((:store
                        [attributes]
                        :collections ((:collection
                                       ([attributes])))
                        :document-types ((:document-type
                                          [attributes]
                                          :elements ((:element ([attributes])))))))))))
```

[attributes] roughly maps to slots from the related classes. Not all slots are supported. Having a long hard look at the classes and the examples will give you a good idea of which slots are supported. If you are still not sure have a look at add-definition-element code for each type of element.

You can add, remove and persist elements of definitions.

Have a look at the api documentation for:

- persist-definition

  A definition is created from the data base object element passed and stored with the data in the appropriate directory for its level.

- add-definition-element

- remove-definition-element

To load a store from a definition you can use load-from-definition.

Have a look at the example and tests/test-definitions.lisp to learn more about how to use definitions.

5 Examples

5.1 Definitions Example

Shows how to create a multiverse from a definition.

```
(ignore-errors (delete-package :naive-examples))

(require 'cl-naive-store.naive-documents)

;; SBCL is idiotic again, it signals an error when compiling a file
;; containing this delete-package form.  You'll have to delete the
;; package yourself between the various examples or tests loads.
(ignore-errors (delete-package :naive-examples))

(defpackage :naive-examples
  (:use :cl :cl-naive-store.naive-documents))

(in-package :naive-examples)

(defun example-location ()
  (cl-fad:merge-pathnames-as-directory
   (user-homedir-pathname)
   (make-pathname :directory (list :relative "multiverse"))))

;;Deleting existing example database
(cl-fad:delete-directory-and-files
 "~/multiverse/universe/simple-store"
 :if-does-not-exist :ignore)

(defvar *multiverse-definition*
  `(:multiverse
    (:name "multiverse"
     :universe-class cl-naive-store.naive-core:universe
     :location ,(example-location)
     :universes

     ((:universe
       (:name "universe"
        :location ,(cl-fad:merge-pathnames-as-directory
                    (example-location)
                    (make-pathname :directory (list :relative "universe")))
        ;;Universe classes can be specific to a universe.
        :universe-class cl-naive-store.naive-core:universe
        :store-class cl-naive-store.naive-documents:document-store
        :collection-class cl-naive-store.naive-documents:document-collection
```

```
:document-type-class cl-naive-store.document-types:document-type
:stores
((:store
  (:name
   "human-resources"
   :collections
   ((:collection
     (:name "laptops"
      :label "Laptops"
      :document-type "laptop"))
    (:collection
     (:name "employees"
      :label "Employees"
      :document-type "employee")))
   :document-types
   ((:document-type
     (:name
      "laptop"
      :label
      "Laptop"
      :elements
      ((:element
        (:name :id
         :label "Serial No"
         :key-p t
         :concrete-type (:type :string)
         :attributes ((:attribute
                       (:name :display
                        :value t))
                      (:attribute
                       (:name :editable
                        :value t)))
         :documentation
               "Unique no that identifies the laptop."))
       (:element
        (:name :make
         :label "Manufaturer"
         :concrete-type (:type :string)
         :attributes ((:attribute
                       (:name :display
```

```
                    :value t))
                  (:attribute
                   (:name :editable
                    :value t)))
     :documentation "Then manufaturer of the laptop."))
   (:element (:name :model
              :label "Model"
              :concrete-type (:type :string)
              :attributes ((:attribute
                            (:name :display
                             :value t))
                           (:attribute
                            (:name :editable
                             :value t)))
              :documentation "Model of the laptop.")))
  :attributes ()
  :documentation "List of laptops the company owns."))

(:document-type
 (:name
  "child"
  :label
  "Child"
  :elements
  ((:element
    (:name :name
     :label "Name"
     :key-p t
     :concrete-type (:type :string)
     :attributes ((:attribute
                   (:name :display
                    :value t))
                  (:attribute
                   (:name :editable
                    :value t)))
     :documentation "Name of child"))
   (:element
    (:name :sex
     :label "Gender"
     :concrete-type (:type :keyword)
```

```
    :attributes ((:attribute
                  (:name :display
                   :value t))
                 (:attribute
                  (:name :editable
                   :value t))
                 (:attribute
                  (:name :value-list
                   :value (:male :female))))
    :documentation
          "Gender of the child, can only be male or female."))
  (:element
   (:name :age
    :label "Age"
    :concrete-type (:type :number)
    :attributes
          ((:attribute
            (:name :display
             :value t))
           (:attribute
            (:name :editable
             :value t))
           (:attribute
            (:name :setf-validate
             :value
                   (lambda (age)
                     (if (<= age 21)
                         (values t nil)
                         (values nil "Child is to old"))))))
    :documentation "How old the child is")))
  :attributes ()
  :documentation "List of laptops the company owns."))

(:document-type
 (:name "employee"
  :label "Employee"
  :elements
        ((:element
          (:name :emp-
           :label "Employee Number"
```

```
  :key-p t
  :concrete-type (:type :number)
  :attributes ((:attribute
                (:name :display
                 :value t))
               (:attribute
                (:name :editable
                 :value t)))
  :documentation "Unique identifier of employee."))
(:element (:name :name
           :label "Name"
           :concrete-type (:type :string)
           :attributes ((:attribute
                         (:name :display
                          :value t))
                        (:attribute
                         (:name :editable
                          :value t)))
           :documentation "Name of employee"))
(:element
 (:name :sex
  :label "Gender"
  :concrete-type (:type :keyword)
  :attributes
        ((:attribute
          (:name :display
           :value t))
         (:attribute
          (:name :editable
           :value t))
         (:attribute
          (:name :value-list
           :value (:male :female))))

  :documentation
        "Gender of the child, can only be male or female."))
(:element
 (:name :dependents
  :label "Children"
  :concrete-type (:type :list
```

```
                    :spec (:type :document
                           :spec (:type "child"
                                  :accessor (:name))))

  :attributes ((:attribute
                (:name :display
                 :value t))
               (:attribute
                (:name :editable
                 :value t)))
  :documentation "List of the employees children"))
(:element
 (:name :laptop
  :label "Laptop"
  :concrete-type (:type :document
                  :spec (:type "laptop"
                         :collection "laptop-collection"
                         :accessor (:id)))

  :attributes ((:attribute
                (:name :display
                 :value t))
               (:attribute
                (:name :editable
                 :value t)))
  :documentation "Laptop allocated to employee"))
(:element
 (:name :first-born
  :label "First Born Child"
  :concrete-type
        (:type :document
         :spec (:type "child"
                :collection "employees"
                :accessor (:emp-no :dependents :name)))

  :attributes ((:attribute
                (:name :display
                 :value t))
               (:attribute
                (:name :editable
```

```
                                        :value t)))
                          :documentation "List of the employees children")))
               :attributes ()
               :documentation "List of laptops the company owns."))))))))))))

(defparameter *multiverse* nil)

(setf *multiverse* (cl-naive-store.naive-core:load-from-definition
                    nil :multiverse *multiverse-definition* :with-children-p t))
```

Inspect `*multiverse*` to see how the schema was realised.

5.2 Basic Example

In this example uses `naive-core` only, ie. the bare minimum functionality. Documents **are not persisted** in this example. There are use cases where the store can be used as kind of temporary database.

```
(ignore-errors (delete-package :naive-examples))

;;Load to use cl-naive-store
(require 'cl-naive-store)

(defpackage :naive-examples (:use :cl :cl-getx :cl-naive-store.naive-core))
(in-package :naive-examples)

;;Required to correctly initialize lparallel:*kernel*.
(initialize)

;;Deleting existing example database
(cl-fad:delete-directory-and-files
 "~/multiverse/universe/simple-store"
 :if-does-not-exist :ignore)

(let* (;;Create a multiverse.
       (multiverse (make-instance
                    'multiverse
                    :name "multiverse"
                    :location "~/multiverse/" ;Setting the location on disk.
```

```
                :universe-class 'universe))
    ;;Create a universe and add it to the multiverse
    (universe (add-multiverse-element
               multiverse
               (make-instance
                'universe
                :name "universe"
                :multiverse multiverse
                :location "~/multiverse/universe/" ;Setting the location on disk.
                :store-class 'store)))
    ;;Create a store and add it to the universe
    (store (add-multiverse-element
            universe
            (make-instance 'store
                           :name "simple-store"
                           :collection-class 'collection)))

    ;;Create a collection and add it to the store
    (collection (add-multiverse-element
                 store
                 (make-instance 'collection
                                :name "simple-collection"
                                ;;Specifying the key element, else its :key
                                :keys '(:id)))))

;;Add some documents to the collection
(add-document collection (list :name "Piet" :surname "Gieter" :id 123))
(add-document collection (list :name "Sannie" :surname "Gieter" :id 321))
(add-document collection (list :name "Koos" :surname "Van" :id 999))

;;Duplicates are handled by default, so this will not cause a duplicate document
(add-document collection (list :name "Piet" :surname "Gieter" :id 123))

;;Query the collection
(query-data collection :query (lambda (document)
                                (<= (getx document :id) 900))))
```

Output:

You can see that Piet Gieter appears only once, because duplicates are handled.

```
((:NAME "Piet" :SURNAME "Gieter" :ID 123)
 (:NAME "Sannie" :SURNAME "Gieter" :ID 321))
```

To allow duplicates you need to set handle-duplicates-p to nil when adding, persisting and loading the data.

5.3 Basic Example with Persistence

In this example only the bare minimum is used and documents added are **persisted**.

```
(ignore-errors (delete-package :naive-examples))

;;Load to use cl-naive-store
(require 'cl-naive-store)
(defpackage :naive-examples (:use :cl :cl-getx :cl-naive-store.naive-core))
(in-package :naive-examples)

;;Required to correctly initialize lparallel:*kernel*.
(initialize)

;;Deleting existing example database
(cl-fad:delete-directory-and-files
 "~/multiverse/universe/simple-store"
 :if-does-not-exist :ignore)

(let* (;;Create a multiverse.
       (multiverse (make-instance
                    'multiverse
                    :name "multiverse"
                    :location "~/multiverse/" ;Setting the location on disk.
                    :universe-class 'universe))
       ;;Create a universe and add it to the multiverse
       (universe (add-multiverse-element
                  multiverse
                  (make-instance
                   'universe
                   :name "universe"
                   :multiverse multiverse
                   :location "~/multiverse/universe/" ;Setting the location on disk.
```

```
                :store-class 'store)))
      ;;Create a store and add it to the universe
      (store (add-multiverse-element
              universe
              (make-instance 'store
                             :name "simple-store"
                             :collection-class 'collection)))

      ;;Create a collection and add it to the store
      (collection (add-multiverse-element
                   store
                   (make-instance 'collection
                                  :name "simple-collection"
                                  ;;Specifying the key element, else its :key
                                  :keys '(:id)))))

  ;;Add some documents to the collection
  (persist-document collection (list :name "Piet" :surname "Gieter" :id 123))
  (persist-document collection (list :name "Sannie" :surname "Gieter" :id 321))
  (persist-document collection (list :name "Koos" :surname "Van" :id 999))

  ;;Clear the collection, ie unload documents from memory so we can
  ;;show that it has been persisted.
  (clear-collection collection)

  ;;Query the collection, query-data will load the data from file if
  ;;the collection is empty
  (query-data collection :query (lambda (document)
                                  (<= (getx document :id) 900))))
```

Output:

```
((:NAME "Piet" :SURNAME "Gieter" :ID 123)
 (:NAME "Sannie" :SURNAME "Gieter" :ID 321))
```

To see the file created for the data go to ~/multiverse/universe/simple-store/simple-collection/ there you should find a simple-collection.log file and you should see the following in the file

```
(:NAME "Piet" :SURNAME "Gieter" :ID 123)
(:NAME "Sannie" :SURNAME "Gieter" :ID 321)
```

5.3.1 Lazy Loading

In this example `query-data` loaded and persisted (in one step) the data into the collection. To show that the data was successfully persisted and that query will lazy load the data when needed, the collection is cleared and only then queried.

You can explicitly load the collection yourself by using `(load-data collection)`.

5.4 Indexed Example

In this example we extend the basic functionality with indexing.

```
;;Load to use cl-naive-store
(require 'cl-naive-store)
(defpackage :naive-examples (:use :cl :cl-getx :cl-naive-store.naive-core
                             :cl-naive-store.naive-indexed))
(in-package :naive-examples)

;;Required to correctly initialize lparallel:*kernel*.
(initialize)

;;Deleting existing example database
(cl-fad:delete-directory-and-files
 "~/multiverse/universe/simple-store"
 :if-does-not-exist :ignore)

;;Create a class that inherits form indexed-collection-mixin and collection.
(defclass indexded-collection (indexed-collection-mixin collection)
  ())

(let* (;;Create a multiverse.
       (multiverse (make-instance
                    'multiverse
                    :name "multiverse"
                    :location "~/multiverse/" ;Setting the location on disk.
                    :universe-class 'universe))
       ;;Create a universe and add it to the multiverse
       (universe (add-multiverse-element
                  multiverse
                  (make-instance
                   'universe
```

```
                :name "universe"
                :multiverse multiverse
                :location "~/multiverse/universe/" ;Setting the location on disk.
                :store-class 'store)))
      ;;Create a store and add it to the universe
      (store (add-multiverse-element
              universe
              (make-instance 'store
                             :name "simple-store"
                             :collection-class 'indexed-collection)))

      ;;Create a collection and add it to the store
      (collection (add-multiverse-element
                   store
                   (make-instance 'indexed-collection
                                  :name "simple-collection"
                                  ;;Specifying the key element, else its :key
                                  :keys '(:id)
                                  ;; Specifying the elements to set up indexes for.
                                  :indexes '((:name :surname)))))
      (results))

;;Load Collection if it was created before.
;; (load-data collection)

;;Add some documents to the collection

(persist-document collection (list :name "Piet" :surname "Gieter" :id 123))
(persist-document collection (list :name "Sannie" :surname "Gieter" :id 321))
(persist-document collection (list :name "Koos" :surname "Van" :id 999))
(persist-document collection (list :name "Frikkie" :surname "Frikkedel" :id 1001))
(persist-document collection (list :name "Tannie" :surname "Frikkedel" :id 1002))

;;Lookup koos using index values and add it to results
(push
 (index-lookup-values collection (list (list :name "Koos")
                                       (list :surname "Van")))
 results)

;;Lookup Frikkedel using index values and add it to results
```

```
  (push
   (index-lookup-values collection (list :surname "Frikkedel"))
   results)

  ;;Query the collection, query-data will load the data from file if the collection is
  ;;and add it to the results
  (push (query-data collection :query (lambda (document)
                                        (<= (getx document :id) 900)))
        results)

  (reverse results))
```

Output:
;; Hashes will obviously be different for you.

```
((:DESC "Looked up Piet using index-lookup-hash." :VALUE
 (:NAME "Piet" :SURNAME "Gieter" :ID 123 :HASH
  "AB2CBF3C-41AE-4656-BE03-B360BCF6A707"))
(:DESC
 "Koos that we looked up using index-lookup-values and the index values of Koos and Va
 :VALUE
 ((:NAME "Koos" :SURNAME "Van" :ID 999 :HASH
   "693A8B07-046E-4643-8FB2-926A965424BF")))
(:DESC
 "A list of both Frikie and Tannie that we looked up using index-lookup-values and the
 :VALUE
 ((:NAME "Tannie" :SURNAME "Frikkedel" :ID 1002 :HASH
   "44A3BF6E-57C4-40BF-8692-54241564611D")
  (:NAME "Frikkie" :SURNAME "Frikkedel" :ID 1001 :HASH
   "5C7E02C7-3674-45FE-94A9-0B843478BA5E")))
(:DESC
 "Queried all id's <= 900 using query-data. The query will use indexes internally when
 :VALUE
 ((:NAME "Sannie" :SURNAME "Gieter" :ID 321 :HASH
   "D0F597B2-12A2-4E05-971C-65B44F5CD4C4")
  (:NAME "Piet" :SURNAME "Gieter" :ID 123 :HASH
   "AB2CBF3C-41AE-4656-BE03-B360BCF6A707"))))
```

5.5 Naive Documents Example

Using `cl-naive-store.naive-documents` gives you a lot of functionality out of the box, but you need to do more work to set it up right.

```
(ignore-errors (delete-package :naive-examples))

;;Setup to use cl-naive-store
(require 'cl-naive-store)
(defpackage :naive-examples (:use
                             :cl
                             :cl-getx :cl-naive-store.naive-core
                             :cl-naive-store.naive-indexed
                             :cl-naive-store.document-types
                             :cl-naive-store.naive-documents))
(in-package :naive-examples)

;;Required to correctly initialize lparallel:*kernel*.
(initialize)

;;Deleting existing example database
(cl-fad:delete-directory-and-files
 "~/multiverse/universe/simple-store"
 :if-does-not-exist :ignore)

;;Create a data definition for an employee
;;It looks like a lot but dont panic its simple.
(defparameter *employee-document-type*
  '(:document-type (:name "employee"
                    :label "Employee"
                    :elements
                    ((:element
                      (:name :emp-no
                       :label "Employee No"
                       :key-p t
                       :concrete-type :string
                       :attributes ((:attribute
                                     (:name :display
                                      :value t))
                                    (:attribute
                                     (:name :editable
```

```
                                    :value t)))))
                        (:element
                         (:name :name
                          :label "Name"
                          :concrete-type :string
                          :attributes ((:attribute
                                        (:name :display
                                         :value t))
                                       (:attribute
                                        (:name :editable
                                         :value t)))))
                        (:element
                         (:name :surname
                          :label "Surname"
                          :concrete-type :string
                          :attributes ((:attribute
                                        (:name :display
                                         :value t))
                                       (:attribute
                                        (:name :editable
                                         :value t))))))
                       :documentation "This type represents a simple employee master.")))

(let* (;;Create Multiverse
       (multiverse
         (make-instance
          'multiverse
          :name "multiverse"
          :location "~/multiverse/" ;Setting the location on disk.
          :universe-class 'universe))
       ;;Add universe to multiverse
       (universe
         (add-multiverse-element
          multiverse
          (make-instance
           'universe
           :name "universe"
           :multiverse multiverse
           :location "~/multiverse/universe/" ;Setting the location on disk.
           :store-class 'document-store)))
```

```
      (store
        (add-multiverse-element
         universe
         (make-instance (store-class universe)
                        :name "simple-store"
                        :collection-class
                        'cl-naive-store.naive-documents:document-collection)))
      (document-type
        (load-from-definition
           store
           :document-type
           *employee-document-type*
           :with-children-p t))
      (collection (add-multiverse-element
                   store
                   (make-instance (collection-class store)
                                  :name "simple-collection"
                                  :document-type document-type
                                  ;; Not specifying the keys to show
                                  ;; that they are retrieved from the document-type
                                  ;; if if no key is set.
                                  ;; :keys ...
                                  ;; Specifying the elements to set up indexes for.
                                  :indexes '((:name :surname)))))
      ;;Add doc to collection
      (doc (persist-document collection
                             (make-document
                              :store (store collection)
                              :collection collection
                              :document-type "employee"
                              :elements (list :name "Piet" :surname "Gieter"
                                              :emp-no 123))))
      (results))

;;Persist store definition with its document type and collection.
(persist multiverse :definitions-only-p t)

;;Add some documents to the collection

(persist-document collection
```

```
                  (make-document
                   :store (store collection)
                   :collection collection
                   :document-type "employee"
                   :elements (list :name "Sannie" :surname "Gieter" :emp-no 321)))

(persist-document collection
                  (make-document
                   :store (store collection)
                   :collection collection
                   :document-type "employee"
                   :elements (list :name "Koos" :surname "Van" :emp-no 999)))

(persist-document collection
                  (make-document
                   :store (store collection)
                   :collection collection
                   :document-type "employee"
                   :elements (list :name "Frikkie" :surname "Frikkedel" :emp-no 1001

(persist-document collection
                  (make-document
                   :store (store collection)
                   :collection collection
                   :document-type "employee"
                   :elements (list :name "Tannie" :surname "Frikkedel" :emp-no 1002)

;;Look up piet by hash
(push (list :desc "Looked up Piet using index-lookup-hash."
            :value (index-lookup-hash collection (getx doc :hash)))
      results)

;;Lookup koos using index values and add it to results
(push
 (list :desc "Koos that we looked up using index-lookup-values and the index values
       :value
       (index-lookup-values collection (list (list :name "Koos")
                                             (list :surname "Van"))))
 results)
```

```
;;Lookup Frikkedel using index values and add it to results
(push
 (list :desc "A list of both Frikie and Tannie that we looked up using index-lookup-
       :value
       (index-lookup-values collection (list :surname "Frikkedel")))
 results)

;;Query the collection, query-data will load the data from file if
;;the collection is empty, and add it to the results
(push
 (list :desc "Queried all id's <= 900 using query-data. The query will use indexes i
       :value
       (query-data collection :query (lambda (document)
                                       (<= (getx document :emp-no) 900))))
 results)

(reverse results))
```

Output:

```
((:DESC "Looked up Piet using index-lookup-hash." :VALUE
  #S(DOCUMENT
     :UNIVERSE #<UNIVERSE (:NAME "universe" :MULTIVERSE "multiverse" :LOCATION
                           "~/multiverse/universe/" :STORES
                           ("simple-store")) {104C234E83}>
     :STORE #<DOCUMENT-STORE (:NAME "simple-store" :UNIVERSE "universe"
                              :LOCATION #P"~/multiverse/universe/simple-store/"
                              :COLLECTIONS ("simple-collection")) {104C2B6C03}>
     :COLLECTION #<DOCUMENT-COLLECTION (:NAME "simple-collection" :STORE
                                        "simple-store" :LOCATION
                                        #P"~/multiverse/universe/simple-store/simple-c
                                        :DOCUMENT-TYPE "employee" :SHARDS
                                        ("simple-c")) {104C411AC3}>
     :DOCUMENT-TYPE "employee"
     :HASH "B58F9362-BDE0-465C-9013-04BF6541C8F7"
     :ELEMENTS (:NAME "Piet" :SURNAME "Gieter" :EMP-NO 123)
     :CHANGES NIL
     :VERSIONS NIL
     :DELETED-P NIL
     :PERSISTED-P T))
```

```
(:DESC
 "Koos that we looked up using index-lookup-values and the index values of Koos and V
 :VALUE
 (#S(DOCUMENT
     :UNIVERSE #<UNIVERSE (:NAME "universe" :MULTIVERSE "multiverse" :LOCATION
                           "~/multiverse/universe/" :STORES
                           ("simple-store")) {104C234E83}>
     :STORE #<DOCUMENT-STORE (:NAME "simple-store" :UNIVERSE "universe"
                              :LOCATION
                              #P"~/multiverse/universe/simple-store/"
                              :COLLECTIONS
                              ("simple-collection")) {104C2B6C03}>
     :COLLECTION #<DOCUMENT-COLLECTION (:NAME "simple-collection" :STORE
                                        "simple-store" :LOCATION
                                        #P"~/multiverse/universe/simple-store/simple-
                                        :DOCUMENT-TYPE "employee" :SHARDS
                                        ("simple-c")) {104C411AC3}>
     :DOCUMENT-TYPE #<DOCUMENT-TYPE (:NAME "employee" :STORE "simple-store"
                                     :ELEMENTS
                                     (:SURNAME :NAME :EMP-NO)) {104C2B6CA3}>
     :HASH "B0E59C8E-2241-490A-B904-1ABA62FD5C9B"
     :ELEMENTS (:NAME "Koos" :SURNAME "Van" :EMP-NO 999)
     :CHANGES NIL
     :VERSIONS NIL
     :DELETED-P NIL
     :PERSISTED-P T)))
(:DESC
 "A list of both Frikie and Tannie that we looked up using index-lookup-values and th
 :VALUE
 (#S(DOCUMENT
     :UNIVERSE #<UNIVERSE (:NAME "universe" :MULTIVERSE "multiverse" :LOCATION
                           "~/multiverse/universe/" :STORES
                           ("simple-store")) {104C234E83}>
     :STORE #<DOCUMENT-STORE (:NAME "simple-store" :UNIVERSE "universe"
                              :LOCATION
                              #P"~/multiverse/universe/simple-store/"
                              :COLLECTIONS
                              ("simple-collection")) {104C2B6C03}>
     :COLLECTION #<DOCUMENT-COLLECTION (:NAME "simple-collection" :STORE
                                        "simple-store" :LOCATION
```

```
                                        #P"~/multiverse/universe/simple-store/simple-
                                        :DOCUMENT-TYPE "employee" :SHARDS
                                        ("simple-c")) {104C411AC3}>
     :DOCUMENT-TYPE #<DOCUMENT-TYPE (:NAME "employee" :STORE "simple-store"
                                     :ELEMENTS
                                     (:SURNAME :NAME :EMP-NO)) {104C2B6CA3}>
     :HASH "6BF9DC95-5158-4B60-A7E0-929BC2D7684F"
     :ELEMENTS (:NAME "Tannie" :SURNAME "Frikkedel" :EMP-NO 1002)
     :CHANGES NIL
     :VERSIONS NIL
     :DELETED-P NIL
     :PERSISTED-P T)
   #S(DOCUMENT
     :UNIVERSE #<UNIVERSE (:NAME "universe" :MULTIVERSE "multiverse" :LOCATION
                           "~/multiverse/universe/" :STORES
                           ("simple-store")) {104C234E83}>
     :STORE #<DOCUMENT-STORE (:NAME "simple-store" :UNIVERSE "universe"
                              :LOCATION
                              #P"~/multiverse/universe/simple-store/"
                              :COLLECTIONS
                              ("simple-collection")) {104C2B6C03}>
     :COLLECTION #<DOCUMENT-COLLECTION (:NAME "simple-collection" :STORE
                                        "simple-store" :LOCATION
                                        #P"~/multiverse/universe/simple-store/simple-
                                        :DOCUMENT-TYPE "employee" :SHARDS
                                        ("simple-c")) {104C411AC3}>
     :DOCUMENT-TYPE #<DOCUMENT-TYPE (:NAME "employee" :STORE "simple-store"
                                     :ELEMENTS
                                     (:SURNAME :NAME :EMP-NO)) {104C2B6CA3}>
     :HASH "97DC3D26-97A9-4B75-A8A3-36B849CAE347"
     :ELEMENTS (:NAME "Frikkie" :SURNAME "Frikkedel" :EMP-NO 1001)
     :CHANGES NIL
     :VERSIONS NIL
     :DELETED-P NIL
     :PERSISTED-P T)))
 (:DESC
  "Queried all id's <= 900 using query-data. The query will use indexes internally whe
  :VALUE
  (#S(DOCUMENT
     :UNIVERSE #<UNIVERSE (:NAME "universe" :MULTIVERSE "multiverse" :LOCATION
```

```
                        "~/multiverse/universe/" :STORES
                        ("simple-store")) {104C234E83}>
   :STORE #<DOCUMENT-STORE (:NAME "simple-store" :UNIVERSE "universe"
                           :LOCATION
                           #P"~/multiverse/universe/simple-store/"
                           :COLLECTIONS
                           ("simple-collection")) {104C2B6C03}>
   :COLLECTION #<DOCUMENT-COLLECTION (:NAME "simple-collection" :STORE
                                      "simple-store" :LOCATION
                                      #P"~/multiverse/universe/simple-store/simple-
                                      :DOCUMENT-TYPE "employee" :SHARDS
                                      ("simple-c")) {104C411AC3}>
   :DOCUMENT-TYPE "employee"
   :HASH "B94547F2-DEE5-4408-A6DB-459E193E0B22"
   :ELEMENTS (:NAME "Sannie" :SURNAME "Gieter" :EMP-NO 321)
   :CHANGES NIL
   :VERSIONS NIL
   :DELETED-P NIL
   :PERSISTED-P T)
#S(DOCUMENT
   :UNIVERSE #<UNIVERSE (:NAME "universe" :MULTIVERSE "multiverse" :LOCATION
                        "~/multiverse/universe/" :STORES
                        ("simple-store")) {104C234E83}>
   :STORE #<DOCUMENT-STORE (:NAME "simple-store" :UNIVERSE "universe"
                           :LOCATION
                           #P"~/multiverse/universe/simple-store/"
                           :COLLECTIONS
                           ("simple-collection")) {104C2B6C03}>
   :COLLECTION #<DOCUMENT-COLLECTION (:NAME "simple-collection" :STORE
                                      "simple-store" :LOCATION
                                      #P"~/multiverse/universe/simple-store/simple-
                                      :DOCUMENT-TYPE "employee" :SHARDS
                                      ("simple-c")) {104C411AC3}>
   :DOCUMENT-TYPE "employee"
   :HASH "B58F9362-BDE0-465C-9013-04BF6541C8F7"
   :ELEMENTS (:NAME "Piet" :SURNAME "Gieter" :EMP-NO 123)
   :CHANGES NIL
   :VERSIONS NIL
   :DELETED-P NIL
   :PERSISTED-P T))))
```

To change a value for an employee you just set the value using getx. For example lets change Sannie's surname.

```
(let ((sannie (first (index-lookup-values
                      (get-multiverse-element
                       :collection
                       (get-multiverse-element
                        :store
                        *universe* "simple-store")
                       "simple-collection")
                      (list (list :name "Sannie")
                            (list :surname "Gieter"))))))

  (setf (getx sannie :surname) "Potgieter"))
```

Which will give you

```
#S(document
   :STORE #<document-STORE {10172A8A73}>
   :COLLECTION #<document-COLLECTION {1017369EA3}>
   :DATA-TYPE "employee"
   :HASH "68434DF1-A04D-4D33-96F1-89D217A193FD"
   :VALUES (:NAME "Sannie" :SURNAME "Gieter" :EMP-NO 321)
   :CHANGES (:NAME "Sannie" :SURNAME "Potgieter" :EMP-NO 321)
   :VERSIONS NIL
   :DELETED-P NIL
   :PERSISTED-P T)
```

The update values can be found in :changes, and will stay there until the document is persisted or abandoned.

Take note that **getx** will return "Potgieter" now even if the document has not been persisted yet.

5.6 Sharding Example

In this example we will use sharding with document types.

```
(ignore-errors (delete-package :naive-examples))

;;Setup to use cl-naive-store
(require 'cl-naive-store)
```

```
(defpackage :naive-examples (:use
                             :cl
                             :cl-getx :cl-naive-store.naive-core
                             :cl-naive-store.naive-indexed
                             :cl-naive-store.document-types
                             :cl-naive-store.naive-documents))
(in-package :naive-examples)

;;Required to correctly initialize lparallel:*kerel*.
(initialize)

(defparameter *surnames*
  #("Smith" "Johnson" "Williams" "Jones" "Brown" "Davis" "Miller")
  "A vector of surnames.")

(defparameter *countries*
  #("Afghanistan" "Albania" "Algeria" "Andorra" "Angola"
    "Antigua and Barbuda" "Argentina" "Armenia" "Australia" "Austria"
    "Azerbaijan" "Bahamas" "Bahrain" "Bangladesh" "Barbados" "Belarus"
    "Belgium" "Belize" "Benin" "Bhutan" "Bolivia" "Bosnia and Herzegovina"
    "Botswana" "Brazil" "Brunei" "Bulgaria" "Burkina Faso" "Burundi"
    "Côte d'Ivoire" "Cabo Verde" "Cambodia" "Cameroon" "Canada"
    "Central African Republic" "Chad" "Chile" "China" "Colombia" "Comoros"
    "Congo" "Costa Rica" "Croatia" "Cuba" "Cyprus"
    "Czechia" "Democratic Republic of the Congo"
    "Denmark" "Djibouti" "Dominica" "Dominican Republic" "Ecuador" "Egypt"
    "El Salvador" "Equatorial Guinea" "Eritrea" "Estonia"
    "Eswatini" "Ethiopia" "Fiji" "Finland" "France"
    "Gabon" "Gambia" "Georgia" "Germany" "Ghana" "Greece" "Grenada"
    "Guatemala" "Guinea" "Guinea-Bissau" "Guyana" "Haiti" "Holy See"
    "Honduras" "Hungary" "Iceland" "India" "Indonesia" "Iran" "Iraq"
    "Ireland" "Israel" "Italy" "Jamaica" "Japan" "Jordan" "Kazakhstan"
    "Kenya" "Kiribati" "Kuwait" "Kyrgyzstan" "Laos" "Latvia" "Lebanon"
    "Lesotho" "Liberia" "Libya" "Liechtenstein" "Lithuania" "Luxembourg"
    "Madagascar" "Malawi" "Malaysia" "Maldives" "Mali" "Malta"
    "Marshall Islands" "Mauritania" "Mauritius" "Mexico" "Micronesia"
    "Moldova" "Monaco" "Mongolia" "Montenegro" "Morocco" "Mozambique"
    "Myanmar" "Namibia" "Nauru" "Nepal" "Netherlands"
    "New Zealand" "Nicaragua" "Niger" "Nigeria" "North Korea"
    "North Macedonia" "Norway" "Oman" "Pakistan" "Palau" "Palestine State"
```

```
    "Panama" "Papua New Guinea" "Paraguay" "Peru" "Philippines" "Poland"
    "Portugal" "Qatar" "Romania" "Russia" "Rwanda" "Saint Kitts and Nevis"
    "Saint Lucia" "Saint Vincent and the Grenadines" "Samoa" "San Marino"
    "Sao Tome and Principe" "Saudi Arabia" "Senegal" "Serbia" "Seychelles"
    "Sierra Leone" "Singapore" "Slovakia" "Slovenia" "Solomon Islands"
    "Somalia" "South Africa" "South Korea" "South Sudan" "Spain"
    "Sri Lanka" "Sudan" "Suriname" "Sweden" "Switzerland" "Syria"
    "Tajikistan" "Tanzania" "Thailand" "Timor-Leste" "Togo" "Tonga"
    "Trinidad and Tobago" "Tunisia" "Turkey" "Turkmenistan" "Tuvalu"
    "Uganda" "Ukraine" "United Arab Emirates" "United Kingdom"
    "United States of America" "Uruguay" "Uzbekistan" "Vanuatu"
    "Venezuela" "Vietnam" "Yemen" "Zambia" "Zimbabwe")
  "A vector of country names.")

(defparameter *employee-document-type*
  '(:document-type (:name "employee"
                    :label "Employee"
                    :elements
                    ((:element
                      (:name :emp-no
                       :label "Employee No"
                       :key-p t
                       :concrete-type :string
                       :attributes ((:attribute
                                     (:name :display
                                      :value t))
                                    (:attribute
                                     (:name :editable
                                      :value t)))))
                     (:element
                      (:name :name
                       :label "Name"
                       :concrete-type :string
                       :attributes ((:attribute
                                     (:name :display
                                      :value t))
                                    (:attribute
                                     (:name :editable
                                      :value t)))))
                     (:element
```

```
                        (:name :surname
                         :label "Surname"
                         :concrete-type :string
                         :attributes ((:attribute
                                       (:name :display
                                        :value t))
                                      (:attribute
                                       (:name :editable
                                        :value t))))))
                      :documentation "This type represents a simple employee master.")))

;;Cannot go below 202! The tests will fail.
(defparameter *size* 10000)

(defparameter *expected-shard-count* 0)

;;Doing this so you can rerun the code from here onwards.
(progn
  ;;Deleting existing example database
  (cl-fad:delete-directory-and-files
   "~/multiverse/universe/simple-store"
   :if-does-not-exist :ignore)

  (let* ((multiverse
            (make-instance
             'multiverse
             :name "multiverse"
             :location "~/multiverse/" ;Setting the location on disk.
             :universe-class 'universe))
         ;;Add universe to multiverse
         (universe
           (add-multiverse-element
            multiverse
            (make-instance
             'universe
             :name "universe"
             :multiverse multiverse
             :location "~/multiverse/universe/" ;Setting the location on disk.
             :store-class 'document-store)))
         (store
```

```
  (add-multiverse-element
   universe
   (make-instance (store-class universe)
                  :name "simple-store"
                  :collection-class
                  'cl-naive-store.naive-documents:document-collection)))
(document-type
  (make-instance
   'document-type
   :name (getf
          (getf *employee-document-type* :document-type)
          :name)
   :label (getf
           (getf *employee-document-type* :document-type)
           :label)
   :elements (mapcar
              (lambda (element)
                (make-instance
                 'element
                 :name(getf (getf element :element) :name)
                 :key-p (getf (getf element :element) :key-p)
                 :concrete-type (getf (getf element :element) :concrete-type)
                 :attributes (getf (getf element :element) :attributes)))
              (getf
               (getf *employee-document-type* :document-type)
               :elements))))
(collection
  (add-multiverse-element
   store
   (make-instance (collection-class store)
                  :name "simple-collection"
                  :keys '(:emp-no)
                  :document-type document-type
                  ;; Creating shards based on the country that the employee
                  ;; belongs to.  It is a bad example you should not shard on
                  ;; any value that could change in the future!
                  :shard-elements '(:country))))
(emp-country  0)
(emp-surname  0)
(unique-countries (make-hash-table :test 'equalp)))
```

```
(setf *expected-shard-count* 0)

(cl-naive-store.naive-core:add-multiverse-element store document-type)

(persist multiverse :definitions-only-p t)

(unless (data-loaded-p collection)
  ;;Populate the collection.
  (dotimes (emp-no *size*)
    ;; We create employees country per country and loop again if we need more:
    (let ((country  (aref *countries* emp-country))
          (surname  (aref *surnames* emp-surname)))

      (unless (gethash country unique-countries)
        (incf *expected-shard-count*)
        (setf (gethash country unique-countries) country))

      (incf emp-surname)

      ;;We only have only 7 surnames start again on surnames
      (when (<= (length *surnames*) emp-surname)
        (setf emp-surname 0)
        ;;move on to next country
        ;;(setf emp-country (mod (1+ emp-country) (length *countries*)))
        (if (< emp-country (- (length *countries*) 1))
            (incf emp-country)
            (setf emp-country 0)))

      (add-document collection
                    (list
                     :country country
                     :surname surname
                     :name (format nil "Slave No ~A" emp-no)
                     :emp-no emp-no)
                    :handle-duplicates-p nil)))

  ;; Bulk perist documents
  (persist-collection collection))
```

```
;;We are just going to investigate the shards and not do a lot data
;;lookups like in other examples.
(list (list :desc "Count of shards in collection."
            :value (length (shards collection)))
      (list :desc "The count of physical shard files for the collection."
            :value (length (uiop:directory-files
                            (cl-fad:merge-pathnames-as-directory
                             (location store)
                             (make-pathname :directory
                                            (list :relative
                                                  (name collection)))))))))
```

Output:

```
((:DESC "Count of shards in collection." :VALUE 195)
 (:DESC "The count of physical shard files for the collection." :VALUE 195))
```

6 User API

The classes, accessors and generic functions used by the typical user of cl-naive-store. Please have a look at the documentation and examples for help on how to use cl-naive-store. The api documentation only deals with the specifics of individual api functions. If you want to cutsomize cl-naive-store then you need to use the functionality described here in condjuction with the Implementor's Api.

6.1 Naive Core

6.1.1 [generic function] query-definitions (definition &key fn element)

Queries the definition passed for an element or elements.

If an element is supplied limits calling the function to those elements or if no function is supplied just fetches elements of keyword element.

If you need more control use cl-naive-ptrees directly.

6.1.2 [generic function] get-definition-element (element-type parent name)

Fetches a definition of the type element-type by name from the parent definition.

6.1.3 [generic function] add-definition-element (element-type definition element &key name-path)

Adds a definition element to the parent definition.

1. add-definition-element ((element-type (eql :collection)) definition collection &key name-path replace-p)
2. add-definition-element ((element-type (eql :document-type)) definition collection &key name-path replace-p)
3. add-definition-element ((element-type (eql :store)) definition collection &key name-path replace-p)
4. add-definition-element ((element-type (eql :universe)) definition collection &key name-path replace-p)

6.1.4 [generic function] remove-definition-element (element-type definition element-name &key name-path)

Removes a definition element from the definition.

1. remove-definition-element ((element-type (eql :universe)) definition element-name &key name-path)
2. remove-definition-element ((element-type (eql :store)) definition element-name &key name-path)
3. remove-definition-element ((element-type (eql :collection)) definition element-name &key name-path)
4. remove-definition-element ((element-type (eql :document-type)) definition element-name &key name-path)

6.1.5 [function] get-definitions (location definition-type)

Returns persisted definitions for the type (multiverse, universe, store, collection) using the passed object that can be a multiverse, universe, store, collection. If no definition-type is not supplied the definition of the object is returned.

6.1.6 [function] get-definition (location definition-type name &key (error-p t))

Returns a persisted definition for the type (multiverse, universe, store, collection) using the passed object that can be a multiverse, universe, store, collection.

6.1.7 [generic function] instance-from-definition (definition-type definition)

Instanciates an element from the definition for the likes of multiverse, universe, store, collection or document-type.

This method has no knowledge of or ignores the existance of parents and children elements.

1. instance-from-definition ((class (eql 'multiverse)) definition)
2. instance-from-definition ((class (eql 'universe)) definition)
3. instance-from-definition ((class (eql 'store)) definition)
4. instance-from-definition ((class (eql 'collection)) definition)
5. instance-from-definition ((class (eql 'document-type)) definition)

6.1.8 [generic function] load-from-definition (parent definition-type definition &key class with-children-p with-data-p)

Instanciates an element from the definition for the likes of multiverse, universe, store, collection or document-type and loads it into the multiverse using the parent.

Multiverse elements usually have a reference to the parent that needs to be set. For instance a collection will have a reference to its store.

Multiverse elements usually also have child elements that could be instanciated for the relevent element. The choice is left up to the user. If the user does want children to also be instanciated they can supply a complete definition or rely on naive-store persisted definition files to be found and used. Set with-children-p for the required behaviour.

Whether documents (the actual data) is loaded after instanciation is a choice of the user. Use with-data-p to affect the behaviour. Just note that if you load data this way you are forgoing lazy loading.

1. load-from-definition ((multiverse multiverse) (definition-type (eql :universe)) definition &key class with-children-p with-data-p)
2. load-from-definition ((universe universe) (definition-type (eql :store)) definition &key class with-children-p with-data-p)
3. load-from-definition ((store store) (definition-type (eql :collection)) definition &key class with-children-p with-data-p)
4. load-from-definition ((store store) (definition-type (eql :document-type)) definition &key class with-children-p with-data-p)

6.1.9 [generic function] load-from-definition-file (parent definition-type name &key class with-children-p with-data-p)

Loads a definition from a file.

6.1.10 [class] shard

Sharding is when you break the physical file that backs the collection into smaller files based on data elements of a document. An instance of a shard class is used to load the documents belonging to the shard into memory.

1. [accessor] mac

   Mac to identify shard.

2. [accessor] location

   The file path to this shard is stored.

3. [accessor] documents

   Documents belonging to shard stored in an adjustable array.

4. [accessor] status

   Used internally during the loading of the documents in a shard to help with locking.

5. [accessor] lock

   Used internally to do shard specific locking.

6.1.11 [class] multiverse

A multiverse is the top structural container for data. A multiverse contains one or more universes. A multiverse could be viewed as a clustering of clusters of databases.

1. [accessor] name

   Mulitiverse name.

2. [accessor] universes

   List of universes contained by this multiverse.

3. [accessor] universe-class

   The class that should be used to make universes.

   NOTES:

   universe-class is delcaritively specied here because stores are dynamicly created when definition files are loaded. (see store notes for more about this.)

4. [accessor] Location

   The directory path to universes for this multiverse.

6.1.12 [class] universe

A universe contains one or more stores. A universe could be viewed as a cluster of databases.

1. [accessor] multiverse

   The multiverse the universe belongs to.

2. [accessor] name

   Universe name.

3. [accessor] stores

   List of stores contained by this universe.

4. [accessor] store-class

   The class that should be used to make stores.

   NOTES:

store-class is delcaritively specied here because stores are dynamicly created when definition files are loaded. (see store notes for more about this.)

5. [accessor] location

   Directory path to stores of this universe.

6.1.13 [class] store

Document types and their associated collections are organized into groups called stores.

NOTES:

collection-class and document-type-class is delcaritively specied here because they are dynamicly created when definition files are loaded. The alternative would be defmethod hell where the customizer of naive-store would have to implement a whole lot of methods that do exactly what the provided methods do just to be able to be type specific in other methods where it is actually needed. Alternatively meta classes could be used for element-class but that opens another can of worms.

1. [accessor] universe

   The universe the store belongs to.

2. [accessor] name

   Store name.

3. [accessor] collection-class

   The class that should be used to make collections.

4. [accessor] collections

   List of collections represented by this store.

5. [accessor] Location

   The directory path to the document-type files and collection files for this store.

6.1.14 [class] collection

A collection of documents of a specific document-type.

1. [accessor] store

   The store that this collection belongs to.

2. [accessor] name

   The collection name.

3. [accessor] location

   The directory path to where files for this collection are stored.

4. [accessor] shards

   A vector of shards.

   NOTES:

   Originally naive-store used lists but with the re-introduction of sharding, we chose to also introduce the use of lparrallel to speed up many functions and lparrallel has a preference for arrays.

5. [accessor] keys

   Keys need to be set to handle duplicates, the default is :key if :key is not found in the document then duplicates will occur.

   NOTES:

   For collections that use cl-naive-document-type there is a fallback the document-type is checked for keys as well and the collection's keys will be set to the keys set in the document-type elements.

6. [accessor] shard-elements

   shard-elements is a list of document element keywords to use for sharding.

6.1.15 [method] documents ((collection collection))

It is a convenience function to retrieve all documents without having to deal with shards.

Loops over all the shards for a collection to gather all the documents.

6.1.16 [method] getx and (setf getx)

Implements getx for multiverse, universe, store and collection.

This means instead of ([accessor] object) you can use (getx object :[accessor]).

6.1.17 [generic function] status (shard) and (setf status)

Used to monitor shards during loading.

1. (setf status) (new-status (shard shard))
2. status ((shard shard))

6.1.18 [generic function] short-mac (shard)

Return a short string containing a prefix of the MAC.

1. short-mac ((shard shard))

6.1.19 [function] match-shard (filename shards)

Check filename against a list of shards to find the matching shard.

6.1.20 [generic function] get-shard (collection shard-mac &key &allow-other-keys)

Get the shard object by its mac. Shard lookups are done so much that there is no choice but to cache them in a hashtable, but that hashtable needs to be thread safe so using safe functions to get and set.

6.1.21 [generic function] make-shard (collection shard-mac))

make-shard ((collection indexed-collection-mixin) shard-mac)
Creates an instance of a shard using the supplied mac.

6.1.22 [function] document-shard-mac (collection document)

Calculating a mac is expensive so caching shard value macs in a hashtable but that hashtable needs to be thread safe so using safe functions to get and set.

6.1.23 [generic function] query-multiverse (element fn)

Queries the multiverse element passed for an element or elements.

1. query-multiverse ((collection collection) fn)
2. query-multiverse ((store store) fn)

3. query-multiverse ((universe universe) fn)

4. query-multiverse ((multiverse multiverse) fn)

6.1.24 [generic function] get-multiverse-element (element-type parent name)

Fetches an element of the type with matching name.

1. get-multiverse-element ((element-type (eql :universe)) (multiverse multiverse) name)

2. get-multiverse-element ((element-type (eql :store)) (universe universe) name)

3. get-multiverse-element ((element-type (eql :collection)) (store store) name)

6.1.25 [generic function] persist (object &key &allow-other-keys)

1. persist ((multiverse multiverse) &key &allow-other-keys)

   Persists a multiverse definition and not what it contains! Path to file is of this general format /multiverse/multiverse-name.universe.

2. persist ((universe universe) &key &allow-other-keys)

   Persists a universe definition and not what it contains! Path to file is of this general format /multiverse/universe-name/universe-name.universe.

3. persist ((store store) &key &allow-other-keys)

   Persists a store definition and not what it contains! Path to file is of this general format /universe/store-name/store-name.store.

4. persist ((collection collection) &key &allow-other-keys)

   Persists a collection definition and the documents in a collection. Path to file for data is this general format /universe/store-name/collection-name/collection-name.log

6.1.26 [function] persist-collection (collection)

Persists the documents in a collection in the order that they where added.

6.1.27 [generic function] add-multiverse-element (parent element &key persist-p)

Adds an instance of a multiverse element to the parent instance.

1. add-multiverse-element ((multiverse multiverse) (universe universe))
2. add-multiverse-element ((universe universe) (store store))
3. add-multiverse-element ((store store) (collection collection))

6.1.28 [generic function] clear-collection (collection)

Clears documents indexes etc from collection.

6.1.29 [generic function] remove-multiverse-element (parent element)

Removes an instance of a multiverse element from the parent instance.

1. remove-multiverse-element ((store store) (collection collection) &key remove-data-from-disk-p)
2. remove-multiverse-element ((multiverse multiverse) (universe universe) &key)
3. remove-multiverse-element ((universe universe) (store store) &key)

6.1.30 [generic function] load-data (collection &key force-reload-p &allow-other-keys)

Loads the data documents of a collection from file or files if sharding is used. If the data is already loaded it wont reload it.

shard-macs is a list of shard macs to indicate which shards should be used. If no shards are specified all shards will be loaded.

6.1.31 [generic function] ensure-location (object)

Tries to find or build path to cl-naive-store files.

1. ensure-location ((object multiverse))
2. ensure-location ((object universe))
3. ensure-location ((object store))
4. ensure-location ((object collection))

6.1.32 [generic function] data-loaded-p (container &key *allow-other-keys)

Checks if the data is loaded for the container, be it universe , store or collection.

NOTES:

This physically checks each collection's underlying concrete data structure for data. This is done because a collection can be empty and still loaded, thus setting a status when loaded became confusing and could be missed by an over loading method.

If you change the underlying container for (shards collection) or the container for (docutments shard) you have to implement data-loaded-p. Your implementation is expected to physically check for document count > 0 and not some status set. Be smart about it you are not expected to return a count so dont waist time counting just check if there is at least one document in the container.

1. data-loaded-p ((collection collection) &key &allow-other-keys)
2. data-loaded-p ((store store) &key &allow-other-keys)
3. data-loaded-p ((universe universe) &key &allow-other-keys)

6.1.33 [generic function] document-values (document)

Returns a plist of document values.

NOTES:

Exists to ease the compatibility of various implementation functions. Basically it blurs the line between plists and more complex documents like cl-naive-store.naive-documents document struct.

This helps keep the amount of specializations needed down considerably.

1. document-values (document)
2. document-values ((document document))

6.1.34 [generic function] key-values (collection values &key &allow-other-keys)

key-values ((collection collection) values &key &allow-other-keys)

Returns a set of key values from the values of a data document. Checks the collection keys or uses hash.

1. key-values ((collection collection) values &key &allow-other-keys)

6.1.35 [generic function] existing-document (collection document &key shard &allow-other-keys)

Finds any documents with the same key values. This could return the exact same document or a similar document.

If a shard is passed in then the search is limited to that shard.

IMPL NOTES:

This is an essential part of loading and persisting documents, take care when implementing.

6.1.36 [generic function] deleted-p (document)

(setf deleted-p) (value document &key &allow-other-keys))

Indicates if a data document has been marked as deleted.

naive-store writes data to file sequentially and when deleting data documents it does not remove a data document from the underlying file it just marks it as deleted.

6.1.37 [generic function] remove-document (collection document &key shard &allow-other-keys)

remove-document ((collection collection) document &key shard &allow-other-keys)

Removes an document from the collection and its indexes. See add-document.

Supplying a shard saves the function from trying to figure out which shard to remove the document from.

1. remove-document ((collection collection) document &key shard &allow-other-keys)

6.1.38 [generic function] delete-document (collection document &key shard &allow-other-keys))

delete-document ((collection collection) document &key shard &allow-other-keys)

Removes a document from the collection, marks the document as deleted and persists the deleted document to disk.

Supplying a shard saves the function from trying to figure out which shard to remove the document from.

1. delete-document ((collection collection) document &key shard &allow-other-keys)

6.1.39 [generic function] add-document (collection document &key shard &allow-other-keys)

Adds a document to the collection, it DOES NOT PERSIST the change, if you want adding with persistance use persist-document or persist the collection as a whole after you have done your adding.

add-document returns multiple values:

The first returned value is the actual document supplied. The second returned value indicates what action was taken ie. was it added newly or was an exiting document replaced. The third returned value is the replaced document.

NOTES:

In general you should not be calling add-document directly, you should use persist-document. Calling add-document directly is allowed so you can create temporary collections that can be thrown away.

cl-naive-store does not have a update-document function, add-document does both and its behaviour can be complex depending on the key parameters supplied. Also the behaviour can differ for different types of collections. Check the appropriate collection documentation for more details.

Supplying a shard saves the function from trying to figure out which shard to add the document to. During loading of a shard naive-impl:%loading-shard% must be used as the default.

1. add-document ((collection collection) document &key (shard naive-impl:%loading-shard%) (handle-duplicates-p t) (replace-existing-p t) &allow-other-keys)

   None of the following will have an effect if handle-duplicates = nil.

   If a document with the same keys exists in the collection the supplied the existing document will be replaced with the supplied document.

   If you set replace-existing-p to nil then an existing document wont be replaced by the supplied document. Basically nothing will be done.

   Supplying a shard saves the function from trying to figure out which shard to add the document to. During loading of a shard naive-impl:%loading-shard% is used.

6.1.40 [generic function] persist-document (collection document-form &key shard &allow-other-keys)

persist-document ((collection collection) document &key shard (handle-duplicates-p t) delete-p &allow-other-keys)

Traverses the document and composes a list representation that is written to file. If the document is new it is added to the collection.

The shard the document should belong to can be passed in to save the function from trying to establish which shard on its own.

1. persist-document ((collection collection) document &key shard (handle-duplicates-p t) delete-p (file-name nil new-file-p) file-stream dont-add-to-collection-p &allow-other-keys)

6.1.41 [generic function] naive-reduce (collection &key query function initial-value &allow-other-keys)

naive-reduce ((hash-table hash-table) &key query function initial-value &allow-other-keys)

naive-reduce ((list list) &key query function initial-value &allow-other-keys)

Uses query to select data documents from a collection and applies the function to those documents returning the result.

NOTES:

Does lazy loading.

naive-reduce ((collection collection) &key query function initial-value shards &allow-other-keys) naive-reduce :before ((collection collection) &key shards &allow-other-keys)

Lazy loading data.

6.1.42 [generic function] query-data (collection &key query &allow-other-keys)

1. query-data :before ((collection collection) &key shards &allow-other-keys)

   Does lazy loading

2. query-data ((collection collection) &key query shards &allow-other-keys)

3. query-data ((store store) &key collection-name query shards &allow-other-keys)

4. query-data ((hash-table hash-table) &key query &allow-other-keys)

   Returns the data that satisfies the query.

   NOTES:

   Does lazy loading.

   Will only use shards supplied if supplied.

6.1.43 [generic function] query-document (collection &key query &allow-other-keys)

1. query-document :before ((collection collection) &key shards &allow-other-keys)

   Does lazy loading.

2. query-document ((collection collection) &key query shards &allow-other-keys)

3. query-document ((store store) &key collection-name query &allow-other-keys)

4. query-document ((list list) &key query &allow-other-keys)

5. query-document ((hash-table hash-table) &key query &allow-other-keys)

   Returns the first last document found, and any others that satisfies the query

   NOTES:

   Does lazy loading.

6.1.44 [generic function] sanitize-data-file (collection &key &allow-other-keys)

This removes all the deleted data documents and history from a collection. When a collection is loaded only the active documents are loaded. Does this by simply writing those active documents out to a new file and then replacing the old file.

1. sanitize-data-file ((collection collection) &key &allow-other-keys)

6.1.45 [generic function] sanitize-universe (universe &key &allow-other-keys)

Sanitize all collections of a universe. See sanitize-data-file for details.

6.2 Naive Indexed

6.2.1 [global parameter] do-partial-indexing

When this is set to t (which is the default), indexing is done for the individual elements of the indexes as well.

6.2.2 [class] indexed-shard (shard)

1. [accessor] hash-index

   Hash table keyed on document uuid for quick retrieval of an document.

2. [accessor] key-value-index

   Hash table keyed on document key values for quick retrieval of an document. Used when doing key value equality comparisons.

6.2.3 [class] indexed-collection-mixin

Collection extension to add very basic indexes.

1. [accessor] indexes

   List of index combinations. Also indexes members partially if **do-partial-indexing** is t, for example '((:emp-no :surname gender)) is indexed as (:emp-no :surname :gender), (:emp-no :surname), :emp-no, :surname and :gender

6.2.4 [class] indexed-collection

There for convenience. Specialisations are done on the mixin.

6.2.5 [method] make-shard ((collection indexed-collection-mixin) shard-mac)

Extends make-shard to deal with indexed collections.

6.2.6 [method] get-shard ((collection indexed-collection-mixin) shard-mac &key &allow-other-keys)

Extends get-shard to deal with indexed collections.

6.2.7 [generic function] hash (document)

(setf hash) (value document)

Returns the hash identifier for a data document. Data documents need a hash identifier to work with naive-store-indexed. naive-store-indexed will edit the document to add a hash identifier when adding documents to a collection. naive-store-indexed uses a UUID in its default implementation.

6.2.8 [generic function] index-lookup-values (collection values &key shards &allow-other-keys)

index-lookup-values ((collection indexed-collection-mixin) values &key (shards (and naive-impl:%loading-shard% (list naive-impl:%loading-shard%))) &allow-other-keys)

Looks up document in key value hash index. If you are not using document-types then the order of values matter.

Will use shards to limit the lookup to specific shards.

6.2.9 [generic function] index-lookup-hash (collection hash &key shards &allow-other-keys)

index-lookup-hash ((collection indexed-collection-mixin) hash (shards (and naive-impl:%loading-shard% (list naive-impl:%loading-shard%))) &allow-other-keys)

Looks up document in UUID hash index.

1. index-lookup-hash ((collection indexed-collection-mixin) hash &key (shards (and naive-impl:%loading-shard% (list naive-impl:%loading-shard%))) &allow-other-keys)

6.2.10 [generic function] add-index (collection shard document position &key &allow-other-keys)

Adds a document to two indexes. The first uses a UUID that will stay with the document for its life time. The UUID is used when persisting the document and is never changed once created. This allows us to change key values without loosing the identify of the original document.

The second is a key value hash index to be used when looking for duplicate documents during persist. If you are not using document-types the order of the keys in the plist matter. To make sure that you dont muck with the order of values/keys in your plists initialize all the possible value pairs with nil so that way the order is set.

A shard must be supplied.

Position saved to use with future add-document to replace actual documents in the document vector if needed. It is done to speed up add-document.

1. add-index ((collection indexed-collection-mixin) shard document position &key key-values &allow-other-keys)

6.2.11 [generic function] remove-index (collection shard document &key &allow-other-keys)

remove-index ((collection indexed-collection-mixin) shard document &key &allow-other-keys)

Removes a data document from the UUID and key value indexes.

A shard must be supplied.

6.2.12 [method] existing-document ((collection indexed-collection-mixin) document &key(shard naive-impl:%loading-shard%) key-values &allow-other-keys)

6.2.13 [method] add-document ((collection indexed-collection-mixin) document &key (shard naive-impl:%loading-shard%) (handle-duplicates-p t) (replace-existing-p t) (update-index-p t) &allow-other-keys)

Duplicates are not allowed for indexed collections!

If the document has no hash and a document with the same keys exists in the collection the supplied document's hash will be set to that of the existing document. The existing document will then be replaced with the supplied document. This is done to maintain hash consistency of the store.

If you set replace-existing-p to nil then an existing document wont be replaced by the supplied document. Basically nothing will be done.

Indexes will be updated by default, if you want to stop index updates set update-index-p to nil. Just remember that if the document is really \"new\" to the collection the indexes will be updated in any case.

6.2.14 [method] naive-reduce ((collection indexed-collection-mixin) &key index-values query function initial-value)

Extends naive-reduce to be able to take advantage of indexing. Reduce is done on values retrieved by the supplier index.

6.2.15 [method] query-data ((collection indexed-collection-mixin) &key index-values query &allow-other-keys)

Extends query-data to be able to take advantage of indexing. Query is done on values retrieved by the supplier index.

6.3 Document Types

6.3.1 [class] element

A definition of an element of a document.

NOTES:

Elements can reference simple types, a complex document or documents based on other document-types.

naive-store can be used as a hierarchical database or a flat databases or a mix.

1. [accessor] name

   Name of the element. This should be a KEYWORD if you want data portability and some internals might expect a keyword.

2. [accessor] document-type

   The document-type that this element belongs to.

3. [accessor] concrete-type

   A user defined "thing" that defines the type specifics of an element.

4. [accessor] key-p

   Indicates that the element is part of the primary key.

   Can be used for indexing and document comparison. For example when a new document is persisted naive-store-documents checks for documents with the same index value and then updates the existing document.

5. [accessor] attributes A property list of additional element attributes.

6.3.2 [class] document-type

A class that can be use to represent a complex document.

NOTES:

The default implementation of cl-naive-store is unaware of document-types when reading and writing documents to and from file. This was by design, to place as little burden on reading and writing documents. Depending on the use of naive-store a user could customize the reading and writing methods of naive-store to use document-types for validation and file layout specifics.

GUI's like cl-wfx use these to help with generic rendering of user input screens.

See cl-naive-definitions for examples of type definitions to get a feel for the intended use.

1. [accessor] store

   The store that this document-type belongs to.

2. [accessor] name

   String representing a document-type name.

3. [accessor] location

   The directory path to where files for this collection are stored.

4. [accessor] element-class

   The class that should be used to make element documents. NOTES:

   element-class is declaratively specified here because so that elements can be dynamicly created when definition type definitions are read from file. See naive-store-documents for usage examples.

5. [accessor] label

   Human readable/formated short description.

6. [accessor] elements

   Field definitions that represents a data unit.

6.3.3 [generic function] get-attribute (element attribute)

6.3.4 [generic function] get-element (document-type element)

6.3.5 [class] document-type-collection-mixin

Collection extention to make collection of a specific document-type.

1. [accessor] document-type

   The document-type that this collection contains documents of.

6.3.6 [class] document-type-store-mixin

1. [accessor] document-type-class

   The class that should be used to make document-type documents. IMPL NOTES: To deal with customization of document-type.

2. [accessor] document-types

   List of document-types represented by this store's collections.

6.3.7 [method] cl-naive-store.naive-core:query-multiverse ((element element) fn)

6.3.8 [method] cl-naive-store.naive-core:query-multiverse ((collection document-type-collection-mixin) fn)

6.3.9 [method] cl-naive-store.naive-core:query-multiverse ((document-type document-type) fn)

6.3.10 [method] cl-naive-store.naive-core:query-multiverse ((store document-type-store-mixin) fn)

6.3.11 [generic function] get-attribute (element attribute))

Gets an attribute of an element.

1. get-attribute ((element element) attribute)

6.3.12 [generic function] get-element (document-type element)

Gets an element from a document type.

6.3.13 [method] cl-naive-store.naive-core:persist-definition ((document-type document-type))

6.3.14 [method] cl-naive-store.naive-core:persist ((document-type document-type) &key &allow-other-keys)

6.3.15 [method] cl-naive-store.naive-core:persist ((store document-type-store-mixin) &key definitions-only-p (children-p t) &allow-other-keys)

6.3.16 [method] cl-naive-store.naive-core:get-multiverse-element ((element-type (eql :element)) (document-type document-type) name)

6.3.17 [method] cl-naive-store.naive-core:get-multiverse-element ((element-type (eql :document-type)) (store document-type-store-mixin) name)

6.3.18 [method] cl-naive-store.naive-core:get-multiverse-element ((element-type (eql :document-type)) (store store) name)

6.3.19 [method] cl-naive-store.naive-core:add-multiverse-element ((document-type document-type) (element element))

6.3.20 [method] cl-naive-store.naive-core:add-multiverse-element((store document-type-store-mixin) (collection collection))

6.3.21 [method] cl-naive-store.naive-core:add-multiverse-element ((store document-type-store-mixin) (document-type document-type))

6.3.22 [method] cl-naive-store.naive-core:add-multiverse-element :after ((store document-type-store-mixin) (collection document-type-collection-mixin))

6.3.23 [method] cl-naive-store.naive-core:instance-from-definition ((class (eql 'element)) definition)

6.3.24 [method] cl-naive-store.naive-core:instance-from-definition ((class (eql 'document-type)) definition)

6.3.25 [method] cl-naive-store.naive-core:load-from-definition ((document-type document-type) (definition-type (eql :element)) definition &key class with-children-p with-data-p)

6.3.26 [method] cl-naive-store.naive-core:load-from-definition ((store document-type-store-mixin) (definition-type (eql :document-type)) definition &key class with-children-p with-data-p)

6.3.27 [method] cl-naive-store.naive-core:instance-from-definition ((class (eql 'document-type-store-mixin)) definition)

6.3.28 [method] cl-naive-store.naive-core:ensure-location ((object document-type))

6.3.29 [method] cl-naive-store.naive-core:load-from-definition-file (parent (definition-type (eql :element)) name &key class with-children-p with-data-p)

6.4.2 [class] document-store (document-type-store-mixin store)

cl-naive-store.naive-documents specialization of store.

6.4.3 [method] cl-naive-store.naive-core:instance-from-definition ((class (eql 'document-collection)) definition)

6.4.4 [method] cl-naive-store.naive-core:load-from-definition ((store cl-naive-store.naive-documents:document-store) (definition-type (eql :collection)) definition &key class with-children-p with-data-p)

6.4.5 [method] cl-naive-store.naive-core:instance-from-definition ((class (eql 'document-store)) definition)

6.4.6 [struct] document

A basic struct that represents a document object. A struct is used because there is meta data that we want to add to the actual document values and there is additional functionality like being able to know what has changed in the values during updates.

- store = The store that the document comes from.
- collection = The collection that the document comes from.
- document-type = The document type specification that describes this document.
- hash = The hash/UUID that uniquely identifies this document
- elements = The actual key value pairs of the document.
- changes = Is used to store setf values when using getx the preffered accessor for values.
- This helps with comparing of values when persisting.
- versions = older key value pairs that represent older versions of the document
- deleted-p = indicates that the document was deleted.
- persisted-p = indicates that the document has been peristed.

6.4.7 [generic function] hash ((document document))

6.4.8 [generic function] key-values ((collection document-collection) document &key &allow-other-keys)

6.4.9 [generic function] document-values ((document document))

6.4.10 [generic function] existing-document ((collection document-collection) document &key key-values &allow-other-keys)

6.4.11 [generic function] persist-document ((collection document-collection) document &key allow-key-change-p delete-p &allow-other-keys)

persist-document for document-collection is lenient in what it takes as a document, it can be of type document or a plist.

6.4.12 [generic function] persist-document index-values ((collection document-collection) (values document) &key &allow-other-keys)

6.4.13 [generic function] getx ((document document) accessor &key &allow-other-keys)

(setf getx) (value (document document) accessor &key (change-control-p t) &allow-other-keys)

getx for documents knows about some of the internals of an document structue so you can get the collection.

Special accessors:

:hash = document-hash

The convention is to append %% to these accessors, for two reasons. First to show that they are special, accessing meta data not actual values of document. Second to avoid any name classes with actual data members.

- :collection~ = document-collection
- :store~ = document-store or (store collection)
- :universe~ = (universe store)
- :type~ = type
- :elements~ = document-elements
- :changes~ = document-changes

- :versions~ = document-versions
- :deleted-p~ = document-deleted-p

store and universe using getx.

6.4.14 [generic function] digx ((place document) &rest indicators)

(setf digx) (value (place document) &rest indicators)

7 Implementors API

cl-naive-store was designed and written to be completely customisable, but this introduces a whole "class" of api functions that should not be used by the average user of cl-naive-store. The functionality covered by this api is only relevant if you want to customise cl-naive-store.

This api in condjunction with the User Api can be used to customise cl-naive-store.

7.1 Naive Core

7.1.1 [function] map-append (fn &rest lists)

Non distructive version of mapcan.

7.1.2 [function] maphash-collect (fn hash-table &key append-p)

Collects the results of a maphash. Pushes to a list by default, use append-p to append instead. NIL results are not collected.

7.1.3 [function] frmt (control-string &rest args)

Short hand for (format nil ..).

7.1.4 [function] trim-whitespacee (string)

Removes white spaces from a string.

7.1.5 [function] empty-p (value)

Checks if value is null/nil or an empty string..

7.1.6 [function] plist-to-values (values)

Returns the values of a plist.

7.1.7 [function] plist-to-pairs (values)

Returns a list of key value pairs of a plist.

7.1.8 [generic function] make-mac (value &key (key mac-key))

Produces a mac from the value. Mac's should differ for different values.
NOTES:
This is used to create shard filenames.

7.1.9 [global parameter] %loading-shard%

Used during the loading of an individual shard. That way no heavy recursive locking has to be done.

7.1.10 [generic function] gethash-safe (key hash &key lock recursive-p)

(setf gethash-safe) (new-value key hash &key lock recursive-p)
Puts lock around hash get access for those cl implementations that dont have a thread safe hashtable.

7.1.11 [generic function] remhash-safe (key hash &key lock recursive-p)

Puts lock around hash remove access for those cl implementations that dont have a thread safe hashtable.

7.1.12 [global parameter] disable-parallel-p

Depending on the data and how naive-store is used switching of parallel processing could produce better performance. This does not disable parallel loading of shards but it does disable all other parallel processing.

Switching off parallel processing is achieved by ignoring the parallel-p argument of do-sequence when **disable-parallel-p** is t.

So if you are customising cl-naive-store use do-sequence for simple parallel processing or make sure that your customization obeys **disable-parallel-p** where possible.

7.1.13 [function] initialize ()

We cannot fork threads while compilinging systems because this prevents saving a lisp image!!! Instead, we must defer forking threads to when we launch the executable image, and initialize the program.

7.1.14 [macro] do-sequence ((element-var sequence &key index-var (parallel-p nil)) &body body)

Iterates over the sequence applying body. In the body you can use the element-var and/or the index-var if supplied.

If you set parallel-p then the body is executed asyncronously. Asyncronous excecution places restraints on how special variables can be used within the body.

From lparallel documentation:

To establish permanent dynamic bindings inside workers (thread-local variables), use the :bindings argument to make-kernel, which is an alist of (var-name . value-form) pairs. Each value-form is evaluated inside each worker when it is created. (So if you have two workers, each value-form will be evaluated twice.)

do-sequence checks if **disable-parallel-p** is set and if it is ignores parallel-p.

Notes:

Uses loop or lparallel:pdotimes depending on parallel-p value.

To get the best out of do-sequence use the parallel option if the sequence is large (> 1000) or the body is excecution heavy.

7.1.15 [macro] with-file-lock ((path &key interval) &body body)

Get an exclusive lock on a file. If lock cannot be obtained, keep trying after waiting a while.

Source: Giovanni Gigante `https://sourceforge.net/p/cl-cookbook/patches/8/`

7.1.16 [macro] file-to-string (file)

Reads a file and returns the contents as a string.

NOTES: You could achieve the same with with-output-to-string, but now you dont have to worry about supplying a string that can be written to.

7.1.17 [macro] with-open-file-lock ((stream file &key (direction :output) (if-exists :append) (if-does-not-exist :create)) &body body)

Opens a file with a file lock and writes to the file.

7.1.18 [function] write-to-file (file object &key (if-exists :append))

Writes to file using with-open-file-lock.

7.1.19 [function] write-list-items-to-file (file list &key (if-exists :append))

Does not wrap items in ().

7.1.20 [function] write-to-file (file object &key (if-exists :append))

Writes to file using with-open-file-lock.

7.1.21 [function] write-to-stream (stream object)

Writes to stream with fresh-lines.

7.1.22 [function] sexp-from-file (pathname)

(setf sexp-from-file) (new-sexp pathname)
Read sexp from file.

7.1.23 [global parameter] break-on-error-log

Causes a break when logging errors of type :error and :warning.

7.1.24 [function] write-log (location type message)

Writes errors to location.
Different Types are written to different files, :error => error.err :warning => warning.wrn :debug => debug.dbl :log => log.lg
Note:
Not writing stuff to .log files because that is what persist uses!!!.

7.1.25 [global parameter] debug-log-p

Switches debug logging or off for debug-log

7.1.26 [function] debug-log (format-control-string &rest arguments-and-keys)

Used for internal debug-logging.
    arguments-and-keys may end with [:file-p f] [:path p].

7.1.27 [generic function] type-of-doc-element (collection sexp)

Reports if the sexp represents a special form.

1. [generic function] persist-form (collection shard element element-type &key &allow-other-keys)

   persist-form (collection shard document (element-type (eql :document)) &key &allow-other-keys) persist-form (collection shard blob (element-type (eql :blob)) &key &allow-other-keys) persist-form (collection shard reference (element-type (eql :reference)) &key &allow-other-keys) persist-form (collection shard hash-table (element-type (eql :hash-table)) &key &allow-other-keys)

   Convert a document element to its list representation.

   IMPL NOTES:

   specialize element type like this (element-type (eql :blob)). DONT specialize on object type directly because that will break type-of-doc-element. If you specialize element you need to supply your own implementation of type-of-doc-element as well.

2. [generic function] persist-parse (collection shard sexp doc &key &allow-other-keys)

   Transcribes document to list form for persistence.

3. [generic function] persist-delete-document (collection shard document file &key &allow-other-keys))

   Marks document as deleted.

   —-

7.1.28 [function] load-document-reference-collection (universe document-ref)

When documents are persisted to file any document values that are referencing an document in a different collection is first sanitized (just enough info to retrieve the document later from where it is stored).

When documents are read from a file the references need to be converted to documents but for that to happen the collection containing the referenced documents need to be loaded first.

7.1.29 [generic function] find-document-by-hash (collection hash)

Finds the document that matches the hash.

7.1.30 [generic function] type-of-sexp (collection sexp)

Reports if the sexp represents a special form, like a blob or reference.

7.1.31 [generic function] compose-special (collection shard sexp type

&key handle-duplicates-p &allow-other-keys)

compose-special (collection sexp (type (eql :document)) &key (handle-duplicates-p t) &allow-other-keys) compose-special (collection sexp (type (eql :blob)) &key (handle-duplicates-p t) &allow-other-keys) compose-special (collection sexp (type (eql :hash-table)) &key (handle-duplicates-p t) &allow-other-keys) compose-special (collection sexp (type (eql :reference)) &key (handle-duplicates-p t) &allow-other-keys)

Does special processing to compose a specific type of document or element.

handle-duplicates-p is often toggled off to speed up loading of data.

7.1.32 [generic function] compose-parse (collection sexp doc)

Processes document form for compose-document.

7.1.33 [generic function] compose-document (collection shard document-form &key (handle-duplicates-p t) &allow-other-keys)

The loading of documents happens in a two step process. First documents are read with (**read-eval** nil). Then the sexp representing a raw document is processed to compose the required in memory representation.

7.2 Naive Indexed

7.2.1 [generic function] cl-murmurhash:murmurhash ((s uuid:uuid) &key (seed cl-murmurhash:*default-seed*) mix-only)

7.2.2 [generic function] index-values (collection values &key &allow-other-keys)

Returns a set of index values from the values of a data document.

7.2.3 [generic function] push-value-index (collection index-values document &key shard &allow-other-keys)

Uses lists within the key-value-index hash-table to store/group documents that match a key value combination.

On updates of documents could end up with duplicate documents returned by the index lookup. The speed more than makes up for the occasional duplicate for now!

7.2.4 [generic function] remove-value-index (collection shad index-values document &key &allow-other-keys)

Removes a value index.

7.3 Documents

7.3.1 [generic function] naive-impl:type-of-doc-element ((collection document-collection) element)

7.3.2 [generic function] naive-impl:persist-form ((collection document-collection) blob (element-type (eql :blob)) &key root parent &allow-other-keys)

7.3.3 [generic function] naive-impl:persist-form ((collection document-collection) document (element-type (eql :reference-form)) &key root parent &allow-other-keys)

7.3.4 [generic function] naive-impl:persist-form ((collection document-collection) document (element-type (eql :child-document)) &key root parent &allow-other-keys)

7.3.5 [generic function] naive-impl:persist-form ((collection document-collection) document (element-type (eql :document))

7.3.6 [generic function] naive-impl:persist-parse ((collection document-collection) element doc &key root parent &allow-other-keys)

7.3.7 [function] document-values-p (list)

Checks if plist contains :values keyword which would indicate the plist represents an document.

8 Tests

`cl-naive-store.tests.asd` loads tests

To run all the tests.

```
(cl-naive-tests:run)
```

To run a specific test, specify the test suite.

```
(cl-naive-tests:run :suites :test-definitions)
;;or
(cl-naive-tests:run :suites :test-basic)
;;or
(cl-naive-tests:run :suites :test-basic-persisted)
;;or
(cl-naive-tests:run :suites :test-indexed)
;;or
(cl-naive-tests:run :suites :test-naive-documents)
;;or
(cl-naive-tests:run :suites :test-sharding-simple)
;;or
(cl-naive-tests:run :suites :test-sharding-indexed)
```

9 Bench Marks

You will have to load the speed test file manually to run the tests.

All the speed tests basically do is:

1. Inserts a lot (1 mil) of documents into a collection.
2. Persist the collection.
3. Clears the collection and reloads it from persisted file(s).
4. Repeats 1 to 3 with a lot more (usually 10m) documents.

The tests will fail if expected time was exceeded for any task.

Expected time however is very dependant on what the specifications of your computer are. Expected time in tests were doubled to try and cater for variance but will most likely fail on anything that has less than 16gb of ram. This does not mean that the database cannot be used with less ram it just means that performance comes at a cost when there is a lot of data.

Query time is not speed tested in these because it depends heavily on the query. The moment you start using indexes queries are however very fast, simple queries complete in fractions of a second.

```
;;Load test-speed.lisp
(cl-naive-tests:run :suites :test-speed-simple)
;;or
;;Load test-indexed.lisp
(cl-naive-tests:run :suites :test-speed-indexed)
;;or
;;Load test-speed-naive-documents.lisp
(cl-naive-tests:run :suites :test-speed-naive-documents)
;;or
;;Load test-speed-sharding.lisp
(cl-naive-tests:run :suites :test-speed-sharding)
```