An Overview of the HDF5 Library Architecture

The HDF Group

> I don't mean a bricks and mortar architect. I mean an architect as used in the words *architect of foreign policy.* I mean architect as in the creating of systemic, structural, and orderly principles to make something work – the thoughtful making of either artifact, or idea, or policy that informs because it is clear. I use the word information in its truest sense. Most of the word information contains the word *inform,* so I call things information only if they inform me, not if they are just collections of data, of stuff. (Saul Wurman)

This document aims to provide a thorough understanding of the inner workings of the HDF5 library by delving into its underlying principles. It covers the systemic, structural, and orderly aspects that make the library function in a clear and informative manner. By going through this document, one can gain insights into the library's architecture and how to use it efficiently. Additionally, it will provide an overview of the various techniques available to simplify the understanding of the operations of the HDF5 library.

Acknowledgment

This work was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

Revision History

Version Number	Date	Comments
v1	March 8, 2023	Version 1 drafted.
v2	December 15, 2023	Version 2, publicly released. Major edits to most sections.

Contents

List of Figures

List of Listings

Acronyms

EOA End of Address. 86

FSM Free Space Manager. 85, 86

HDF5 Hierarchical Data Format, v5. 13

1. Introduction

This document is intended to provide a detailed and in-depth understanding of how the HDF5 library operates. It delves into the underlying principles of the library and examines its systemic, structural, and orderly aspects responsible for its clear and informative functioning. Anyone going through this document can gain valuable insights into the library's architecture and learn how to use it efficiently. Furthermore, it provides an overview of various techniques that can be used to simplify the understanding of the operations of the HDF5 library, making it easier to work with and more accessible for users of all levels of expertise.

1.1. Audience and Goals

This document aims to help readers achieve the outcomes they want. Such readers might be

- Community contributors looking for guidance in addressing issues
- New library developers trying to find their bearings in a large code base
- HDF5 library extension developers who want to "dig deeper"
- HDF5 power users who want to understand "how the library really works."

While comprehensiveness (= covering a wide range of topics related to a subject) is one of the goals for this document, completeness (= having all necessary parts) is not, at least not for this version. While helping readers achieve outcomes is the purpose of this document, it is not focused on specific tasks and is not a "cookbook."

We have organized the material in two parts: a set of guided tours (Chapter 2) and a reference (Chapter 4). The guided tours provide specific contexts where subsets of architectural concerns can be discussed. The reference is "context-free" in the sense that each topic can be consulted and stands on its own.

1.2. Architecture

Architectural documentation of a system or software consists of a set of models for an audience. Different models represent different perspectives or views of, in this case, the HDF5 library. One of the perhaps simplest models of the HDF5 library is shown in Figure 1.1.

Despite its simplicity, a black box model gives us context for other models and lets us ask relevant questions. One of its central premises is that by correlating stimuli and responses, we might be able to draw certain conclusions about the black box's "inner workings." However, it is not straightforward to correlate HDF5 API calls with actual IO activity. This is the case for three reasons:

1. The HDF5 library is an intermediate abstraction layer

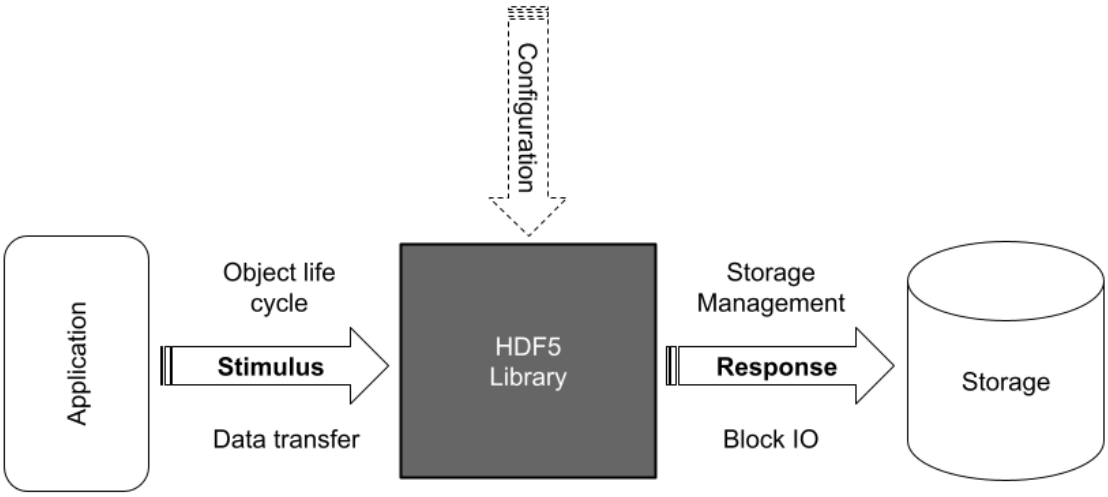


Figure 1.1. – A black box model of the HDF5 library.

2. It (de-)serializes collections of complex object constellations into storage structures
3. IO activity occurs in blocks.

Hence, we must open the box, which is what this document is about.

The stimuli to which we can subject the black box come in two forms: object life cycle management instructions and data transfer requests. Object life cycle management means tasks such as creating and modifying familiar HDF5 objects (datasets, groups, datatypes). By data transfer, we mean any kind of `*read*` or `*write*` API call, which is intended to transfer user (meta-)data into storage.

The black box responds to stimuli in two forms: storage management and block IO operations. Typical examples of storage include file systems and object stores. Specific responses depend on how HDF5 data model primitives are encoded in storage primitives. Examples include single- or multi-file layouts or collections of storage objects.

When observing the stimulus/response “traffic" of even simple applications, the correlation is much weaker and less direct than we might expect and sometimes outright counter-intuitive (e.g., an input operation triggering an output operation). Unlike a mathematical function, which consistently produces the same outputs for the same inputs, the relation between stimuli and responses exhibited by the HDF5 library is non-functional, albeit deterministic. This is due to the HDF5 library’s role as a translator between HDF5 model idioms, represented in memory, and storage idioms. To be effective and efficient, and to bridge the speed differential between memory and storage, the HDF5 library maintains a state. This state dilutes the simple stimulus/response logic and manifests itself in what might appear as a delayed response or an aggregate response to a sequence of stimuli.

The architecture of a system or software is usually presented in models that fall under different perspectives or views, such as the ones shown in Table 1.1.

Perspective or View	Description
Purpose/objective	What the client wants
Form	What the system is
Behavioral or functional	What the system does
Performance objectives or requirements	How effectively the system does it
Data	The information retained in the system and its relationships
Managerial	The process by which the system is constructed and managed

Table 1.1. – Six common perspectives or views. [16]

For a given system or software, not all of them are equally important or relevant at all times. For example, the library build process and testing (managerial view) is covered in the first guided tour, whereas the modular library structure (functional view) is the organizing principle for the reference.

1.3. Purpose, Objectives, and Values

The purpose of the HDF5 library is to ensure efficient and equitable access to science and engineering data stored in HDF5 files across platforms and environments, now and forever. Toward that purpose, the two main objectives are:

1. Self-describing data
2. Portable encapsulation and access

Self-describing data captures all information about itself necessary to reproduce and interpret it as intended by its producer. A storage representation must preserve the self-describing nature when transferring such representations over a network or to different storage. At the same time, it should be accompanied by a portable library that allows applications to access the data without knowing anything about the details of the representation.

The "marriage" [14] of the HDF5 file format and library is a specific implementation of the primitives and operations defined by the HDF5 data model and was adapted for several specific use cases over time.

The implementation is guided by the following values, which are listed in no particular order.

Extensibility The degree to which users can change behavior and appearance.

- Datatypes, conversions
- Filters
- Links
- Data virtualization
- Storage types
- File format micro-versioning

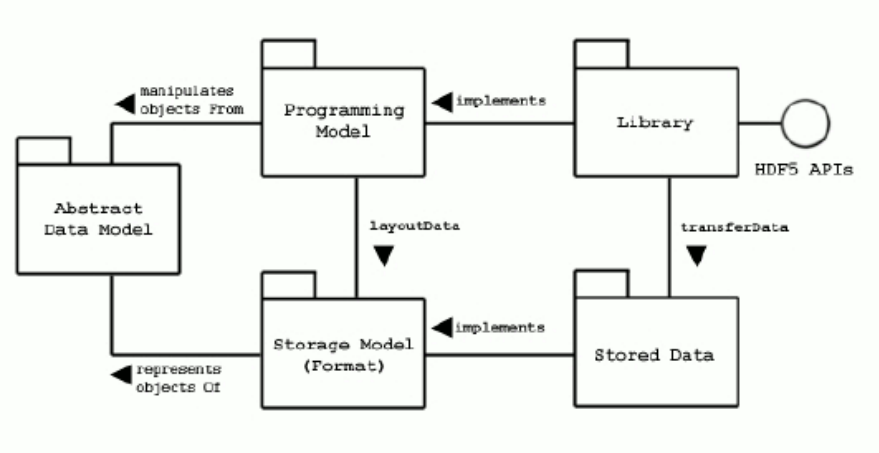


Figure 1.2. – Models

Compatibility, Longevity, & Stability Things that worked before continue to work the same indefinitely

- Quasi-static data model
- Backward- and forward compatibility
- API Versioning
- Open file format specification

Efficiency Effective operation as measured by comparing production with cost (as in time, storage, energy, etc.)

- Algorithmic complexity
- Scalability

Maintainability The degree to which it can be modified without introducing fault

- Additive software construction [12]

Progressiveness A measure of eagerness to make progress and leverage modern storage technology

Freedom Specifically, free software means users have the four essential freedoms [7]

- The freedom to run the program as you wish, for any purpose (freedom 0).
- The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies to help others (freedom 2).
- The freedom to distribute copies of your modified versions to others (freedom 3). By doing this, you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

1.4. How this document is organized

This document has three parts. Part one is a set of guided tours that are used to uncover architectural elements, but without going into too much detail. The second part, currently a fragment, is meant as a reflection on problems in the HDF5 library architecture. Part three is a reference section, where specific architectural elements are explored in greater detail, in some cases, for the first time, in non-source code form.

1.5. Typographic conventions

bold headers, titles

italic emphasis

`monospace` source code, package names, configuration file settings, file system paths and folder names.

2. Guided Tours

These guided tours introduce certain architectural elements in a specific context, and the level of detail is calibrated to the context. Complete coverage of the architectural concepts is provided in Chapter 4.

A simple program typically sets the course of a specific guided tour, and the call stack is the starting point for "HDF5 library stratigraphy." We will examine how changes in the program change the call stack and activate and exercise different parts of the library. We hope the samples extracted from several well-placed "stratigraphic test wells" will create a reasonably accurate and complete picture of the library's modular structure and functional allocation.

2.1. Meet the HDF5 library

Throughout this tour, we describe the run-time use of the term 'HDF5 library', which provides complete coverage of the architectural concepts in different contexts with different meanings. It should be clear from the context which meaning is intended, but we list them here for completeness. The term 'HDF5 library' refers to one of the following:

1. An architectural concept.
2. A body of source code, versions of which can be found on GitHub [11].
3. A binary compiled and configured for a particular platform using CMake [13] or Autotools [6].
4. An HDF5 library binary loaded into the context of a process, a library *instance*.
5. The internal state of an HDF5 library instance.
6. A combination of one or more of the above, in a generic sense.

We begin with a 35,000-foot overview of how the HDF5 library is structured. As shown in Figure 2.1, *the primary purpose of the HDF5 library is to provide mappings between in-memory representations of HDF5 objects or entities and storage representations*. There are many possible implementations of such mappings, all of which share the HDF5 data model and library API. The API comprises two types of functions: functions dealing with the management of virtual objects and functions configuring API behavior or virtual object support functions. In Figure 2.1, this division is represented by the boxes labeled 'Virtual Object Layer' and 'Non-VOL API Calls.'

The Virtual Object Layer (VOL) is an HDF5 library extension interface for implementing actual mappings of virtual objects to storage and implementing the operations associated with such objects' life cycle (creation, deletion, copying, etc.). Modules implementing such mappings are called VOL *connectors* and are typically packaged as dynamically loadable plugins. In Figure 2.1, three VOL connector examples are shown: the so-called 'Native VOL', the DAOS VOL [20], and the Async VOL [21]. Notice that certain VOL connectors are stackable and that the native VOL is *not* implemented as a plugin. The latter is due to the VOL-centric

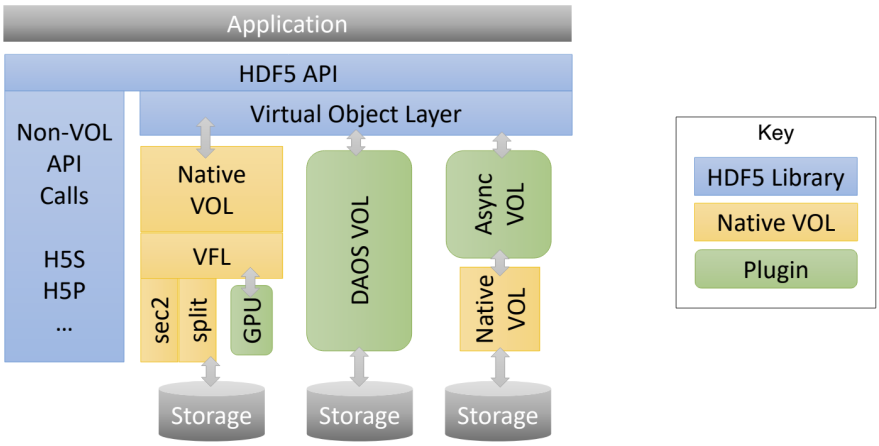


Figure 2.1. – The VOL architecture.

library architecture being relatively new. It was introduced in HDF5 library release 1.12.0 and underwent a significant revision for the HDF5 1.14.0 release. Before introducing the VOL layer, the HDF5 library was more or less identical to the native VOL connector (plus the non-VOL support functions). Perhaps the main takeaway from Figure 2.1 is that what's now considered the HDF5 library is a lot smaller than expected. Today, the HDF5 library consists of an API, the VOL interface, and a non-VOL API management infrastructure. Of course, that bare-bones library doesn't do anything useful, and we need at least one VOL connector that implements an actual mapping of HDF5 virtual objects to storage.

Since, during this tour, we only examine library initialization and finalization, we will not encounter any "proper" VOL functions. No virtual objects are constructed or mapped to storage, and no operations are performed on them. We will witness the initialization and finalization of the library's non-VOL infrastructure and the native VOL packages. During subsequent tours, we will see the native VOL in action, i.e., mapping HDF5 virtual objects to files formatted according to the HDF5 file format specification [10] via the native VOL's Virtual File Layer (VFL) extension interface.

Unless stated otherwise and to keep the discussion simple, in this chapter, we include the native VOL connector when we refer to the HDF5 library. As we saw, however, this is not entirely true.

In this section, we describe the life cycle of an HDF5 library instance running in the context of an application process. This includes lazy library initialization, dynamic configuration, and finalization. At the end of this section, the reader should have enough information to answer the following questions:

- What does it mean for a library instance to be initialized/finalized?
- What triggers a library instance initialization/finalization?
- What's the sequencing of library initialization/finalization?
- As a library developer, what library changes require careful consideration of/coordination with library initialization/finalization?

Prototype. Consider the HDF5 version of a “Hello, World!" program shown in Listing 2.1. The naming `H5[open,close]()` is perhaps a little misleading (Why not `H5[init,finalize]()`?), but, for reasons which will become clear later, most users and developers will never call these two functions directly.

```
1 #include "common.h"
2 int main() {
3     H5open();
4     printf("Hello, HDF5!\n");
5     H5close();
6     return 0;
7 }
```

Listing 2.1 – Explicit HDF5 library run-time initialization and finalization.

Goals. The main goals of library initialization and finalization are:

- Setup and tear-down of error handling infrastructure
- Module initialization and finalization, including
  - (De-)Registration of API entity types (or classes)
  - Initialization and finalization of global variables
  - Retrieval of environment variables, e.g., for the discovery of plugin paths

Control flow. A cursory inspection of the call graph of the program shown in Listing 2.1 suggests that `H5[open,close]()` proceed directly to `H5_[init,term]_library()`, respectively.

`H5_init_library()` is responsible for calling the initializers of all non-VOL and native VOL library modules (aka “packages") in the correct order so that no module is initialized before its prerequisite modules. A table in `H5_init_library()` establishes the order of initialization. The table, as implemented at the time of this writing, is shown in Listing 2.2.

Most package initializers (the `H5*_init*` routines) register identifier classes or check environment variables. The only notable exception is the `H5T` package initializer, which initializes identifiers (stored in global variables) for built-in datatypes and several pre-defined datatype transformations. For this tour, where we do not encounter any identifiers, it is sufficient to know that identifiers are ephemeral handles of API type `hid_t`, which the library uses to track API entities. For efficiency and error checking, the identifier space is “partitioned” into identifiers of different types, and developers can create (register) custom identifier types to track their entities.

As the table shows, a few modules (`H5P`, `H5VL`) require phased initialization. `H5_default_vfd_init()`, in connection with `H5P_init_phase2()`, registers and initializes the POSIX VFD as the default file driver. Finally, `H5VL_init_phase2()` initializes the remaining modules according to a similar table shown in Listing 2.3.

`H5VL_init_phase2()` concludes the HDF5 library initialization.

Only `H5[A,D,F,G,L,M,T]` API functions will pass through the VOL layer; all other packages support non-VOL API calls or library internal infrastructure.

```
struct {
    herr_t (*func)(void);
    const char *descr;
} initializer[] = {
    {H5E_init, "error"}
,   {H5VL_init_phase1, "VOL"}
,   {H5SL_init, "skip lists"}
,   {H5FD_init, "VFD"}
,   {H5_default_vfd_init, "default VFD"}
,   {H5P_init_phase1, "property list"}
,   {H5AC_init, "metadata caching"}
,   {H5L_init, "link"}
,   {H5S_init, "dataspace"}
,   {H5PL_init, "plugins"}
/* Finish initializing interfaces that depend on the interfaces above */
,   {H5P_init_phase2, "property list"}
,   {H5VL_init_phase2, "VOL"}
};
```

Listing 2.2 – `H5_init_library()` initialization table.

```
struct {
    herr_t (*func)(void);
    const char *descr;
} initializer[] = {
    {H5T_init, "datatype"}
,   {H5O_init, "object header"}
,   {H5D_init, "dataset"}
,   {H5F_init, "file"}
,   {H5G_init, "group"}
,   {H5A_init, "attribute"}
,   {H5M_init, "map"}
,   {H5CX_init, "context"}
,   {H5ES_init, "event set"}
,   {H5Z_init, "transform"}
,   {H5R_init, "reference"}
};
```

Listing 2.3 – `H5VL_init_phase2()` initialization table.

```c
#include "common.h"
int main() {
  unsigned int majnum, minnum, relnum;
  H5get_libversion(&majnum, &minnum, &relnum);
  printf("Hello, HDF5 %u.%u.%u!\n", majnum, minnum, relnum);
  return 0;
}
```

Listing 2.4 – Implicit HDF5 library initialization.

At this point, the library would be considered initialized with its default configuration (native VOL + POSIX VFD), and the application can proceed to create and manipulate HDF5 virtual objects that are mapped to HDF5 files in a POSIX-compliant file system. For other mappings, different VOL connectors or VFD plugins would be required.

The library has a default behavior to shut down when the application exits by registering an `atexit(3)` handler, `H5_term_library()`. This will invoke modules' `H5*_[top]_term_package()` functions. The shutdown table is too extensive to be repeated here. It is also heavily commented on and should be consulted for more in-depth information. Akin to the phased initialization for certain modules, the finalization of a few modules is split into "top" and "bottom" portions. Note that some entries in the shutdown table are marked as "barriers," and if a new module should only be shutdown *strictly after* the preceding modules, then it should be marked as a barrier. This is achieved by setting the `wait` parameter of the `TERMINATOR` macro for the corresponding module to `true`.

Data flow. The library's initialization and finalization state are tracked in two global variables, namely `H5_INIT_GLOBAL` and `H5_TERM_GLOBAL`. These variables are specifically set by two library functions `H5_init_library()` and `H5_term_library()`, respectively. The initialization and finalization of library modules typically involve registering and unregistering module-specific identifier types (for example, `H5I_DATASPACE_CLS` and `H5I_SPACE_SEL_ITER_CLS` for `H5S`) and the initialization/finalization of module-specific global variables. Perhaps the most elaborate example is `H5T`, which requires the initialization/finalization of built-in datatypes such as `H5T_IEEE_F64BE_g` and conversion functions such as `H5T__conv_double_ldouble()` in the global variable `H5T_g`. Otherwise, there is very little data flow activity during library initialization/finalization.

Variants. In Listing 2.4, a variant of the "Hello, World!" example is shown. The notable difference is the new function `H5get_libversion()`, while the previous version had calls to `H5open()` and `H5close()`, which have been removed in the new version. It turns out that the HDF5 library initializes itself as needed whenever an application either enters the library through an API function call such as `H5get_libversion()`, or when an application evaluates an HDF5 symbol that represents a macro such as `H5F_ACC_RDONLY` or `H5F_ACC_RDWR`, a property-list class identifier such as `H5P_FILE_ACCESS`, a VFD identifier such as `H5FD_FAMILY` or `H5FD_SEC2`, or a type identifier such as `H5T_NATIVE_INT64`.

The library provides a couple of macros that initialize the library as necessary. Library initialization is checked as a side-effect of the `FUNC_ENTER_API*` macros used at the top of most API functions. HDF5 library

symbols other than functions are provided through `#defines` that use the `H5OPEN` macro to introduce a library-initialization call (`H5open()`) at each site where a non-function symbol is used.

Two noteworthy variants of HDF5 library initialization and finalization will be discussed in later tours. One is the case of MPI-parallel HDF5 , where multiple library instances simultaneously operate on one or more files with different sharing patterns. The other one is the use case of a library built with thread-safety enabled (see section 2.7.1). In both cases, the changes to HDF5 library initialization and finalization are “around the edges" and deal with MPI initialization, finalization, and API lock management, respectively.

Developer considerations. When a developer exports a new symbol as part of the HDF5 library, they should ensure that an application cannot enter the library in an uninitialized state through a new API function or read an uninitialized value from a non-function HDF5 symbol.

If a developer adds a module to the library that must be initialized with the rest of the library, then they should insert its initializer into the right place in the table.

If a developer adds a module that needs to release resources during library shutdown, they should add a call to the shutdown table at the right place.

Source code recommended for study.

- `H5_[init,term]_library()` in `H5.c`
- `H5E_[init,term]_package()` in `H5E.c`
- `H5T_init(), H5T_top_term_package()` in `H5T.c`
- `H5VL_init_phase2()` in `H5VLint.c`

2.1.1. Summary

Given the substantial library infrastructure we have seen during this tour, it is too easy to “miss the architecture for the code". The key takeaway from this tour is that, as suggested in Figure 2.1, the HDF5 library consists of

1. A (C-) API
2. An extension interface (VOL) to implement mappings between HDF5 virtual objects and storage representations
3. A supporting infrastructure (non-VOL) for API handle management, API properties, etc.
4. A default mapping (native VOL) between HDF5 virtual objects and the content of files formatted according to the HDF5 file format specification [10].

Library initialization and finalization are usually triggered by API entry and an `atexit(3)` handler. The HDF5 library source code is organized modularly, and initialization and finalization proceed through modules in a defined order, respecting inter-module dependence. Developers must guard against uncontrolled API entrance, violations of module interdependence during library initialization and shutdown, and failures to release unused resources.

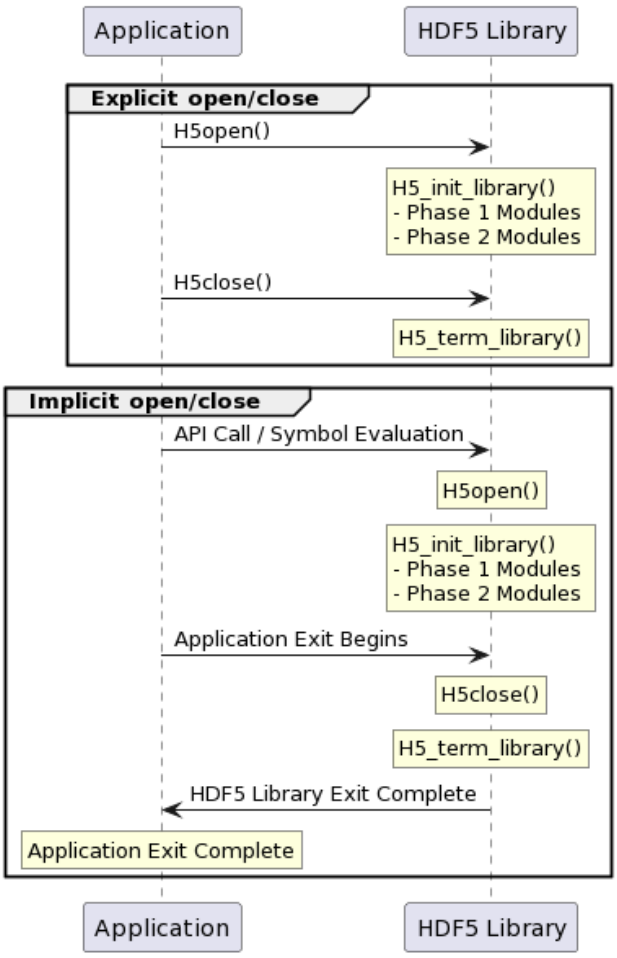


Figure 2.2. – Library initialization/termination sequence diagram.

2.2. The life cycle of HDF5 files

Throughout this tour, we describe the life cycle of HDF5 files.

Terminology. As we saw in the previous tour (Section 2.1), the term “HDF5 library" has several context-dependent meanings. Likewise, the term ‘HDF5 file’ refers to one of the following:

1. A kind of HDF5 virtual object.
2. A portion of the HDF5 library that represents an open HDF5 file.
3. A terminal VOL connector-specific in-storage representation, including “file-less."
4. One or more files stored in a file system or object store formatted according to the HDF5 file format specification [10].
5. A combination of one or more of the above, in a generic sense.

```c
#include "common.h"
int main() {
    hid_t file_id = H5Fcreate("hello.h5", H5F_ACC_TRUNC, H5P_DEFAULTx2);
    H5Fflush(file_id, H5F_SCOPE_GLOBAL);
    size_t size;
    H5Fget_filesize(file_id, &size);
    printf("File size: %zu\n", size);
    H5Fclose(file_id);
    return 0;
}
```

Listing 2.5 – An “empty” (800 B) HDF5 file.

We first describe an HDF5 file’s runtime life cycle before covering a few finer points in subsequent sections.

2.2.1. Life cycle

In this section, we describe the internals of an HDF5 file’s life cycle, including VOL connector and VFD selection and handling, by reviewing the `H5Fcreate()`, `H5Fopen()`, and `H5Fclose()` stacks. At the end of this section, the reader should have enough information to answer the following questions:

- How do HDF5 library API calls get “routed" through the VOL layer?
- Under what circumstances do they enter the native VOL connector?
- How are HDF5 files represented in memory?
- What is the HDF5 virtual file layer (VFL), and how does it relate to the rest of the HDF5 library?
- How does the HDF5 library select virtual file drivers (VFD)?

Prototype. Let’s consider the simplest program to create a new HDF5 file shown in Listing 2.5.

We create an “empty" HDF5 file. Of course, such a minimal HDF5 file must contain a root group! Finally, we print its in-storage size. (As we will see later, the call to `H5Fflush()` is necessary to get an accurate size.)

Goals. The main goals of the HDF5 file creation/open/close life cycle are:

- Create an in-memory representation of an HDF5 file.
- Return a file identifier or handle (`hid_t`) to the caller.
- Serialize and persist the file object in storage.
- Free all resources associated with the file upon closure of the last handle.

Control flow. As we saw in Section 2.1, any API call, in this case, `H5Fcreate()`, will trigger library initialization, which includes the registration of the native VOL as the default VOL connector and the POSIX VFD (sec2) as the default file driver.

The call sequence for each API routine used in Listing 2.5 may be found in Figures: 2.3 (`H5Fcreate()`), 2.4 (`H5Fflush()`), 2.5 (`H5Fget_filesize()`), and 2.6 (`H5Fclose()`). For modules whose API functions are routed through the VOL layer, we have adopted the notion of top and bottom portions, e.g., `H5F Top` and `H5F Bot`. Since the HDF5 library modules predate the introduction of the VOL layer, the code is not neatly separated into VOL handling/native VOL code portions. We refer to a module's VOL handling code portions as the module's 'top' portion and the native VOL portion as the module's 'bottom.'

`H5Fcreate()` enters the VOL layer at `H5VL_file_create()` via `H5F__create_api_common()`. This routine handles sanity checking and calling the internal VOL layer routine, `H5VL_file_create()`. The native VOL connector is selected, and its file creation callback, `H5VL__native_file_create()`, is invoked. The open routine from the file module, `H5F_open()`, does all the heavy lifting in terms of allocation and initialization of various in-memory structures (see the section on data flow), creates a metadata cache for the file under construction (`H5AC_create()`), and culminates in file superblock and root group creation.

On the following line, `H5Fflush()` is routed through the VOL layer via `H5F__flush_api_common()`, which invokes the `H5VL_file_specific()` callback with operation type `H5VL_FILE_FLUSH`. The flush operation is handled by the native VOL's `H5VL__native_file_specific()`.

Similarly, the call to `H5Fget_filesize()` is routed through the VOL layer's `H5VL_file_optional()` callback with the operation type `H5VL_NATIVE_FILE_GET_SIZE`, eventually reaching the native VOL's implementation of the get filesize operation within `H5VL__native_file_optional()`. This invokes the native VOL's internal `H5F__get_max_eof_eoa()`, which invokes the VFL's `H5FD_get_eo[a,f]()` and adds them to the file's base address obtained from `H5FD_get_base_addr()`.

`H5Fclose()` proceeds to decrement the file ID's reference count via `H5I_dec_app_ref()`. If the reference count has reached 1, which it has in this example, this triggers the invocation of the free function for the particular identifier type, a file ID (`H5I_FILE`), in this case. The free function in this case is `H5F__close_cb()` via which we re-enter the VOL through `H5VL_file_close()`, which in turn calls the native VOL's `H5VL__native_file_close()`. This triggers the release of in-memory file resources, including flushing and destroying the file-specific metadata cache, via `H5F__dest()`.

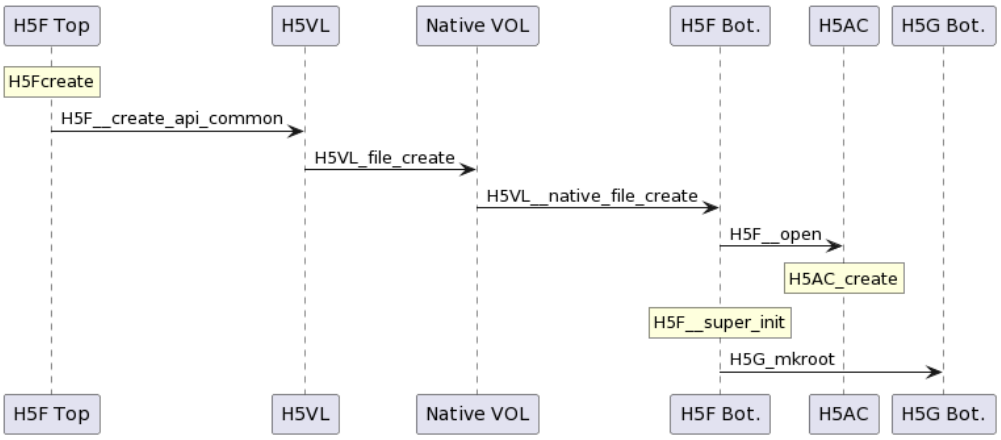


Figure 2.3. – File creation sequence diagram

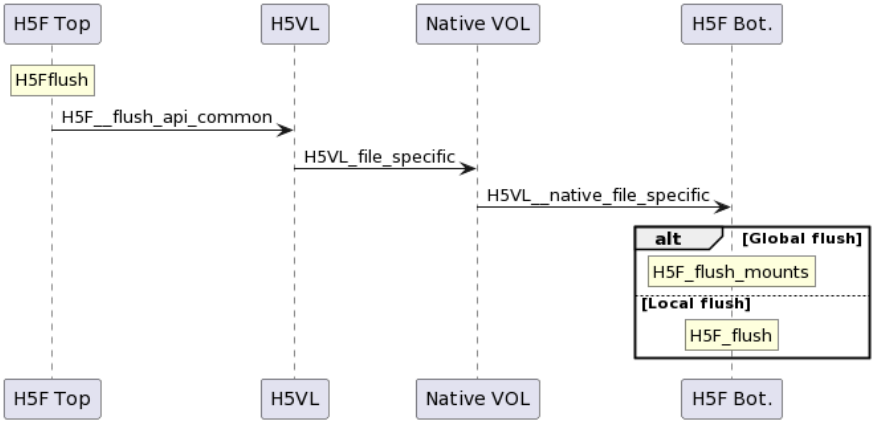


Figure 2.4. – File flush sequence diagram

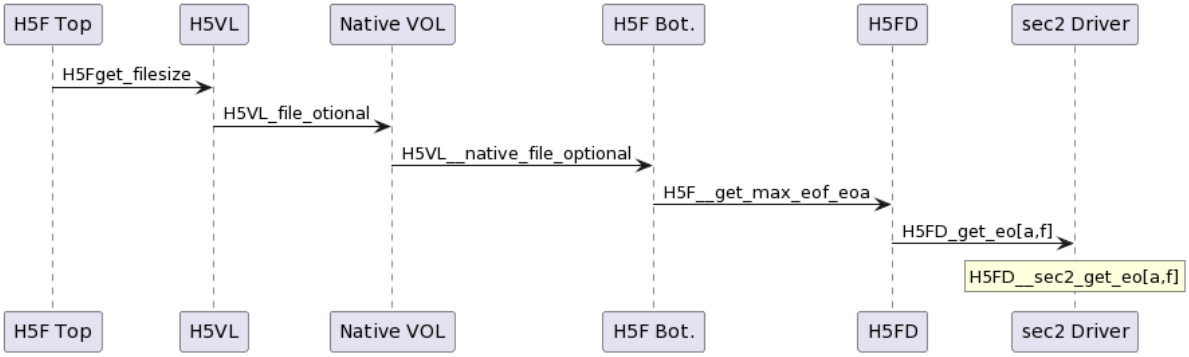


Figure 2.5. – Get filesize sequence diagram

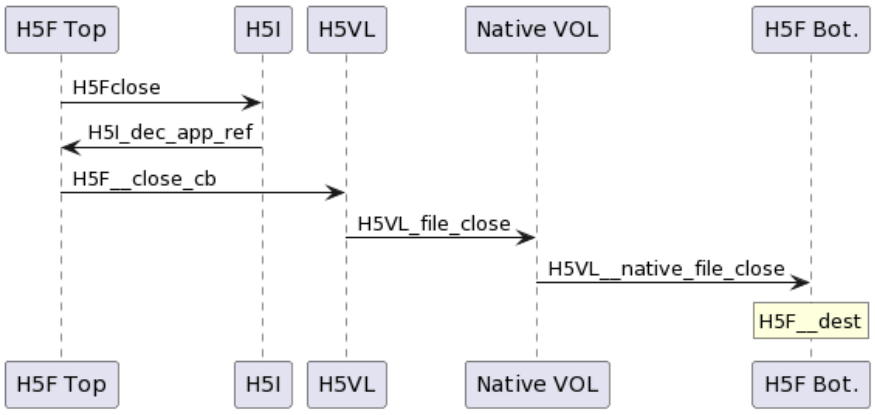


Figure 2.6. – File close sequence diagram

```
struct H5F_t {
    char           *open_name;
    char           *actual_name;
    H5F_shared_t   *shared;
    H5VL_object_t  *vol_obj;
    unsigned        nopen_objs;
    H5FO_t         *obj_count;
    bool            id_exists;
    bool            closing;
    struct H5F_t   *parent;
    unsigned        nmounts;
};
```

Listing 2.6 – A top-level file descriptor.

Before looking at variants of this control flow, we look at the data flow and provide the “Cliff notes" for the VOL and VFL layers. A detailed architectural overview can be found in Section 4.1.16.

Data flow. Every time `H5F[create,open]()` is called, a structure of type `H5F_t` (see Listing 2.6) is allocated.

Since HDF5 files can be opened multiple times with different `hid_t` handles in the same process, the in-memory file representation is split into `H5open()` instance-specific top (`H5F_t`) and shared bottom portions of type `H5F_shared_t`. Both `H5F_t` and `H5F_shared_t` are defined in `H5Fpkg.h`. `H5F_shared_t` is too extensive to be reproduced here, but it contains all the information that is common among `H5Fopen()` instances, including:

- VOL connector and VFD information
- The metadata cache for this file and its configuration
- Version information
- Open HDF5 objects in the file
- File space management configuration
- Various status information.

We strongly encourage the reader to review the well-documented `H5F_shared_t` definition!

Notice that most fields in `H5F_shared_t` show the file system heritage and represent information related to persisting HDF5 files in a file system according to the file format specification [10]. For example, the `sblock` field of `H5F_shared_t` (see Listing 2.7) matches almost verbatim the file format specification.

This is our first time encountering the metadata cache (`H5AC` module). For now, it is essential to remember that there is one metadata cache instance for each HDF5 file in the sense of `H5F_shared_t`, and `H5F_super_t` is its first protected entry.

Until this point, everything happened in memory, and no “real" I/O has occurred. Considering the overhead associated with frequent small I/O operations, this behavior is quite desirable. Ignoring “raw" data I/O, metadata, such as the file superblock, will be written only on flush events, or when an HDF5 file (`H5F_shared_t`)

```
typedef struct H5F_super_t {
    H5AC_info_t cache_info;
    unsigned    super_vers;
    uint8_t     sizeof_addr;
    uint8_t     sizeof_size;
    uint8_t     status_flags;
    unsigned    sym_leaf_k;
    unsigned    btree_k[H5B_NUM_BTREE_ID];
    haddr_t     base_addr;
    haddr_t     ext_addr;
    haddr_t     driver_addr;
    haddr_t     root_addr;
    H5G_entry_t *root_ent;
} H5F_super_t;
```

Listing 2.7 – File superblock in-memory representation.

is closed, and the associated metadata cache instance is destroyed (see `H5F__dest()`). Speaking roughly, metadata I/O happens less frequently as the direct consequence of API calls than as the result of metadata cache events, such as entry evictions.

The downside of this apparent de-correlation of in-memory metadata management and I/O adds a certain fragility to the state of an HDF5 file that's open in a modifying access mode. If an application terminates unexpectedly, the library may not get to persist modified portions of the file state. Future implementations might offer better protection against file corruption.

It is hard to overstate the central role of the metadata cache's role in the HDF5 native VOL, although that might not yet be apparent. We will have more to say about the metadata cache in subsequent tours.

2.2.2. Detour: The Virtual Object Layer

Goals The purpose of the Virtual Object Layer (VOL) is to decouple the HDF5 library's API and object abstractions from a particular storage implementation while requiring minimal change in how the API is used. It allows for independent, potentially user-defined VOL connectors to handle virtual object operations. A common use case for VOL connectors is to change how HDF5 files are stored, mapping an HDF5 file in memory to something other than a POSIX file in storage - for example, a collection of objects in an AWS S3 bucket. The VOL layer's capabilities are more general than just changing the implementation of storage - a connector can perform arbitrary operations (such as logging, caching, or mirroring) whenever a routine that is part of the VOL API is used.

VOL Architecture Figure 2.1 shows how the VOL fits into the library's structure. Operations that deal with storage or virtual objects are part of the VOL API, which means that calls pass through one or more VOL connectors before reaching storage. There are two types of VOL connectors: pass-through connectors, which pass control flow to another connector after performing some operation, and terminal connectors, which do not pass on to another connector and are generally used to map objects to storage.

Library functions which are a part of the VOL API internally call a function of the form `H5*__<operation>_api_common()`, which routes the call to the VOL layer via `H5VL_<operation>()`. The VOL layer determines which connector to use by checking the property list that was passed as an argument to the original API call.

For example, with the default file access property list (FAPL) specified by `H5P_DEFAULT`, the top level `H5Fopen()` internally calls `H5F__open_api_common()`, which passes control to `H5VL_file_open()`. Because the default FAPL specifies the native VOL connector, `H5VL_file_open()` will invoke the native VOL's file open callback `H5F_open()` (note the underscore!), which opens an HDF5 file object from a locally-stored POSIX file. A VOL connector provides pointers to its callbacks in its `H5VL_class_t` struct. (See Section 4.1.16.)

VOL connectors other than the native connector are distributed as plugins loaded into an application. There are three ways to load a connector: dynamic loading using environment variables, dynamic loading with API calls, and manual linking to the connector.

Dynamic Loading with Environment Variables Dynamic loading with environment variables does not require rebuilding or modifying the application. However, it does not allow an application to load multiple VOL connectors.

At runtime, the library checks the environment variable `HDF5_VOL_CONNECTOR` for the name of a VOL connector to load. This name is case-sensitive and is specified manually by each connector - the DAOS VOL connector, for example, expects to be discovered under the name `daos`.

If the VOL connector is not installed to a default system path, then the environment variable `HDF5_PLUGIN_PATH` must be used to specify the absolute path to the connector. Generally, this will be the `/lib` subdirectory of wherever the connector was installed.

If the VOL connector is successfully loaded through environment variables, the library's default property lists, accessed via the `H5P_DEFAULT` macro, will tell the VOL layer to use the dynamically loaded connector.

Dynamic Loading via API calls There are two methods to load a VOL connector as a plugin and assign it an `hid_t` handle: `H5VL_register_by_name()` and `H5VL_register_by_value()`. When choosing to use `H5VL_register_by_value()`, a VOL connector identifier needs to be provided, which is a unique number assigned to each registered VOL by the HDF Group. On the other hand, `H5VL_register_by_name()` is a method that generally requires the same name that would be provided to `HDF5_VOL_CONNECTOR`, although it may vary depending on the connector.

Once the connector has been registered and assigned an `hid_t` handle, this handle may be provided to `H5Pset_vol` to modify a FAPL, telling any VOL layer function that checks this FAPL to use the supplied connector.

Manual Linking Linking a VOL connector requires modifying the source and build of the target application but provides more flexibility in how the connectors may be used.

The requirements to link and use a connector are as follows:

- Link to the connector's library at application build time
- Include the connector's public headers in the application

- Potentially invoke any initialization/termination functions of the connector at the start/end of the application

Once these requirements are satisfied, the FAPL is modified using `H5Pset_fapl_<vol_name>()` or `H5Pset_<vol_name>()` so that any VOL API calls provided with that FAPL will use the specified connector. Note that the exact function name for this operation will vary by connector in a way that does not adhere to any strict pattern.

2.2.3. Detour: The Virtual File Layer

A popular metaphor refers to HDF5 files as “file systems in a file." The Virtual File Layer (VFL) adds some credibility to this metaphor. This section reviews the HDF5 Virtual File Layer, its requirements and assumptions, and its place in the library.

Overview The VFL serves a purpose similar to the VOL – outsourcing the implementation of specific storage-related operations to potentially user-defined Virtual File Drivers (VFDs) while minimizing the changes necessary in user applications. However, the scope of the VFL is narrower: it defines a mapping between the HDF5 address space and file system-like storage.

As seen in Figure 2.1, the Virtual File Layer (VFL) sits at the bottom of the native VOL. It is a native VOL extension interface. If a non-native VOL is used, the VFL will be bypassed entirely. Just as the native VOL is the default VOL connector, the sec2 driver (or ’POSIX VFD’) is the default VFD, which uses POSIX functions to perform unbuffered I/O (with respect to the HDF5 library) to a single file. (See Section 4.1.8 for a configuration in which the HDF5 library performs buffering.)

Using a VFD. Unlike the VOL layer, which has only the native VOL built into the library, several VFDs are included with the library and can be enabled with build options. These include the MPI-IO VFD for parallel file access and the read-only S3 (ROS3) VFD, which allows read-only access to HDF5 files stored as objects in AWS S3 buckets.

The process of using a VFD that is not built with the library is similar to the process of using a manually linked VOL connector:

- The VFD must be linked to the application at the application’s build time
- The VFD’s public headers must be included

After meeting the necessary requirements, the VFD can be enabled by modifying the FAPL using either `H5Pset_driver_by_name()` or `H5Pset_driver_by_value()`. `H5Pset_driver_by_name()` requires a case-sensitive name specific to each VFD, and `H5Pset_driver_by_value()` requires a ‘driver value,’ a unique number assigned to each VFD that is registered with The HDF Group.

Alternatively, each VFD should define a function with a name of the form `H5P_set_fapl_<VFD_name>()`, which also modifies a FAPL to specify the use of that VFD. Listing 2.9 shows a function of this form being used to enable the core VFD, a VFD that manages HDF5 files as contiguous memory buffers and maps I/O operations to memory accesses.

```c
#include "common.h"
int main() {
    hid_t fapl_id = H5Pcreate(H5P_FILE_ACCESS);
    H5Pset_libver_bounds(fapl_id, H5F_LIBVER_LATEST, H5F_LIBVER_LATEST);
    hid_t file_id = H5Fcreate("my_file.h5", H5F_ACC_TRUNC, H5P_DEFAULT,
        fapl_id);
    H5Fflush(file_id, H5F_SCOPE_GLOBAL);
    size_t size;
    H5Fget_filesize(file_id, &size);
    printf("File size: %zu\n", size);
    H5Fclose(file_id);
    H5Pclose(fapl_id);
    return 0;
}
```

Listing 2.8 – A modern "empty" (195 B) HDF5 file.

VFL Architecture. Recall from the section on VOL Architecture that the property list for an operation determines which VOL is used. If the native VOL is selected, the property list also contains information about which file driver to use. Continuing the earlier example, the top level `H5Fopen()` eventually calls the native VOL's `H5F_open()`, which invokes the VFL's `H5FD_open()`. This VFL function uses the 'open' callback of the active file driver, which for the default sec2 driver is `H5FD__sec2_open()`. This driver callback is where the actual work to open a file in memory is performed.

Each file driver provides pointers to its callbacks in an `H5FD_class_t` struct. These callbacks are invoked by handler functions with names of the form `H5FD_<operation>()` in the VFL layer.

Variants. Signaling the presence of architecture (!), the two variants shown in Listings 2.8 and 2.9 follow the same sequence that's shown in Figures 2.3 through 2.6. They diverge from the prototype (Listing 2.5) in only two minor technical details.

The code shown in Listing 2.8 will modify the default values of the `[low,high]_bound` fields of the file's `H5F_shared_t` structure. This affects how certain file metadata is serialized (a process triggered by metadata cache events) if a newer file format specification offers better alternatives or a use case requires it.

As described earlier, the code in Listing 2.9 instructs the library to select a non-default VFD. This will update the `lf` field of the file's `H5F_shared_t` structure, which wraps the VFD-specific callbacks. In other words, only `H5FD_*()` calls will be affected by this change.

2.2.4. Summary

During this tour, we saw the interplay between the API, the Virtual Object Layer, the native VOL connector, and the Virtual File Layer. We saw how and which API calls are "routed" through the VOL layer, which is perhaps the most important takeaway. We got a glimpse of how HDF5 files are represented in memory. Along the way, we encountered the metadata cache as one of the central architectural elements of the current library implementation. There is one metadata cache instance per HDF5 file, and rather than being directly controlled by API calls, metadata I/O occurs around metadata cache events indirectly triggered by API calls. While

```c
#include "common.h"
int main() {
    hid_t fapl_id = H5Pcreate(H5P_FILE_ACCESS);
    H5Pset_fapl_core(fapl_id, 4096, 1);
    hid_t file_id = H5Fopen("my_file.h5", H5F_ACC_RDWR, fapl_id);
    size_t size;
    H5Fget_filesize(file_id, &size);
    printf("File size: %zu\n", size);
    H5Idec_ref(file_id);
    H5Idec_ref(fapl_id);
    return 0;
}
```

Listing 2.9 – Using a different VFD.

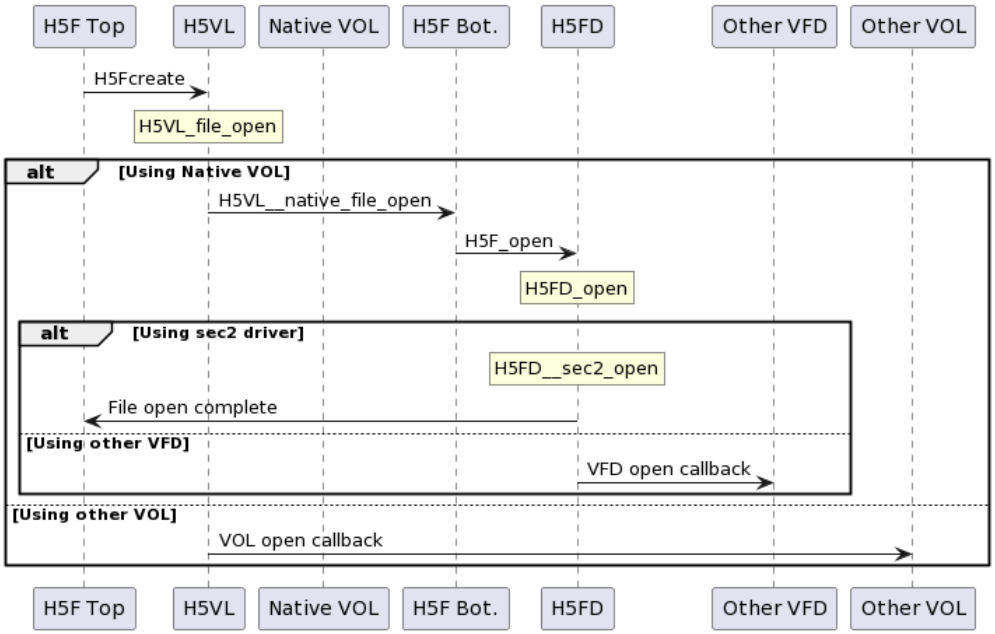


Figure 2.7. – File open through VOL and VFL sequence diagram

improving overall I/O performance, the “split nature" of the state of a modified open HDF5 file makes it more vulnerable to unexpected application termination.

With extension interfaces such as VOL and VFD, the HDF5 library’s extensibility is apparent but may make the simple creation of an HDF5 file appear overly complicated. However, very few applications stop at creating empty HDF5 files, and we saw some of the benefits of this when examining variants of the empty HDF5 file creation prototype.

During the next tour, we will examine how the HDF5 library handles user (meta-)data.

2.3. What is the HDF5 library doing to my data?

In the section ‘IV.B. Disk Format: Level 2B - Data Object Data Storage’, the HDF5 file format specification states the following regarding the storage of array values (from datasets or attributes):

> Multi-dimensional array data is stored in C order; in other words, the “last" dimension changes fastest.
>
> Data whose elements are composed of atomic datatypes are stored in IEEE format unless they are defined explicitly as being stored in a different machine format with the architecture-type information from the datatype header message. This means that each architecture will need to [potentially] byte-swap data values into the internal representation for that particular machine.
>
> Data with a variable-length datatype is stored in the global heap of the HDF5 file. Global heap identifiers are stored in the data object storage.
>
> Data whose elements are composed of reference datatypes are stored in several different ways depending on the particular reference type involved. Object pointers are just stored as the offset of the object header being pointed to, with the size of the pointer being the same number of bytes as offsets in the file.
>
> Dataset region references are stored as a heap ID, which points to the following information within the file heap: an offset of the object pointed to, number-type information (same format as header message), dimensionality information (same format as header message), sub-set start and end information (in other words, a coordinate location for each), and field start and end names (in other words, a [pointer to the] string indicating the first field included and a [pointer to the] string name for the last field).
>
> Data of a compound datatype is stored as a contiguous stream of the items in the structure, with each item formatted according to its datatype.

Having read through pages and pages of HDF5 file format intricacies, many readers may find this statement near the end of the document [10] to sound anticlimactic. Undoubtedly, the writer of these lines must have run out of time or money, and he or she tried to cut corners because this sounds too simple to be true! Well, it is accurate, and the answer to the question, “What is the HDF5 library doing to my data?" is “Not much." or “Not what you might be thinking." Borrowing networking terminology, the HDF5 library is *framing* user data rather than manipulating the data itself. (Here, we ignore specific use cases such as datatype conversion or compression.) Similar to TCP/IP protocol layers (see Figure 2.8), there are multiple framing layers in the HDF5 file format.

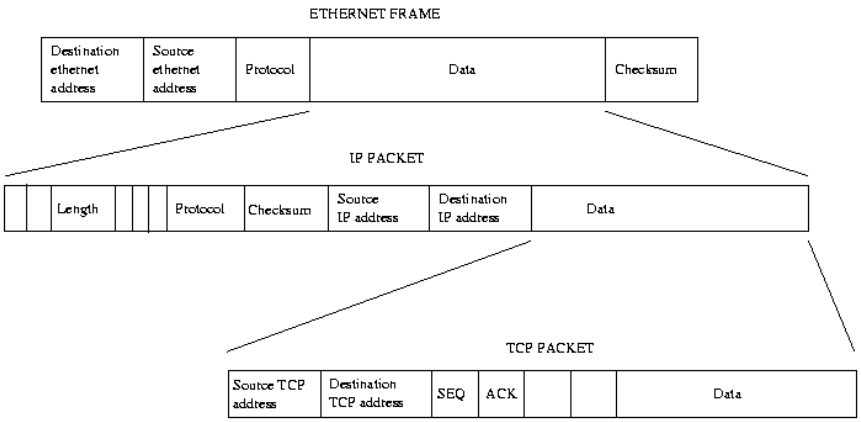


Figure 2.8. – TCP/IP Protocol Layers[18]

In this section, we explore how the native VOL performs this “framing" of user (meta)data. After clarifying the difference between user metadata and user data, we will see why and how it treats them separately. At the end of this tour, the reader should be able to explain Figure 2.16 to their colleagues!

2.3.1. User metadata and data

Most applications using HDF5 are there to solve a scientific, engineering, or business problem. Users of the application think in terms of the particular application domain, such as particle velocities, sensor readings, or stock prices. Be that as it may, the application’s creator has adopted a convention or profile on representing these domain objects using HDF5 data model entities. Like a mathematical model, in the broadest sense, the HDF5 data model is about variables and the relationships between variables. These variables, or datasets, are array variables (their values are multidimensional arrays), and relationships are expressed by arranging them in graph-like structures with labeled edges or links. Furthermore, datasets and arrangements, or groups, can be decorated by “small” (in the domain sense) named array variables, so-called attributes. With the “grammar" of how these arrangements, or (logical) files, can be put together, this is essentially all there is to the HDF5 data model. In this context, we refer to dataset values (arrays!) as *user data*, and everything else is considered *user metadata*. In other words, user data is the “heavy-weight" or problem-scale data, proportional to the number of degrees of freedom, the resolution, the spatio-temporal extent, etc. On the other hand, user metadata describes user data, e.g., the unit of the underlying variable, the calibration or sampling rate of an instrument, the time step, etc. Note that dataset metadata, such as rank, extent, element type, etc., is also metadata.

For example, the HDF5 data model instance created by the program in Listing 2.10 would consist of the array `[0,0,0,0,0,0,0,0,0,0]` as its user data and metadata such as

- The HDF5 file contains a single array variable linked as `data` in the root group.
- The element type of the array variable is “integers between -2,147,483,648 and 2,147,483,647."
- The rank of the array variable is one, and its extent is ten elements.

```c
#include "common.h"
int main() {
    hid_t file_id = H5Fcreate("data.h5", H5F_ACC_TRUNC, H5P_DEFAULTx2);
    hid_t fspace = H5Screate_simple(1, (hsize_t[]){10}, NULL);
    hid_t dset = H5Dcreate(file_id, "data", H5T_NATIVE_INT, fspace,
        H5P_DEFAULTx3);
    H5Dclose(dset);
    H5Sclose(fspace);
    H5Fclose(file_id);
    return 0;
}
```

Listing 2.10 – Dataset life cycle.

The HDF5 library lets us persist this information in a file system's physical file `data.h5`. In Figure 2.16, we show the major library components involved in the framing of this information and suggest that user metadata and data are treated differently.

Following the Callgrind profile of Listing 2.10, a call sequence for `H5Dcreate()` similar to the one shown in Figure 2.9 emerges. Notice that we have skipped the VOL layer incantation, which applies to `H5Dcreate()`, and we have suppressed a few details about object header composition and metadata cache entry pinning, etc. The gist of the call sequence is this:

1. As a "side-effect" of linking the newly minted dataset to the root group, we create a dataset object header conforming to the file format specification [10].
2. The object header is placed into the file's metadata cache, and the corresponding entry is marked as dirty. That way, it will get flushed/written to the file when the metadata cache is destroyed (`H5AC_dest()`), which happens in the wake of `H5Fclose()`.
3. Finally, the freshly minted dataset's file address and link name are added to the root group's symbol table, and the B-tree index is updated.

This sequence covers the left portion of Figure 2.16. What happened to the array? If you follow the Callgrind profile closely, you will notice that no array ever gets created and written. For one, there is no statically or dynamically allocated array in Listing 2.10. However, the actual reason is the default behavior of the library that prescribes so-called late allocation for datasets with contiguous layouts. Space allocation in the file is delayed until an element of the dataset is modified, which triggers the allocation and initialization with fill values for elements that aren't modified. In other words, nothing happens in the right portion of Figure 2.16 for this simple example. It's a pure metadata affair.

The dataflow for datasets is similar to files in that the in-memory representation has top (`H5D_t`) and bottom (`H5D_shared_t`) portions. Both are defined in `H5Dpkg.h`. The top portion is unique to each instance of an opened dataset, and the bottom portion is a shared struct that is only created once for a given dataset. (The dataset's object header is constructed from the information in the bottom portion.) Thus, if a dataset is opened twice, there will be two handles (IDs) and two `H5D_t` structs, both sharing one `H5D_shared_t`.

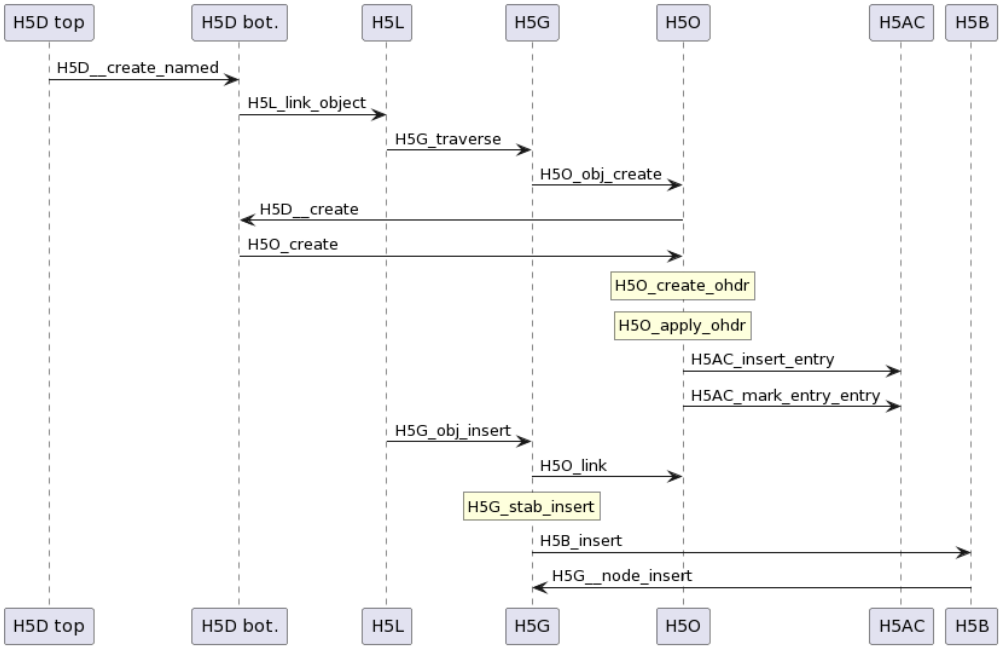


Figure 2.9. – Process to create a contiguous dataset.

2.3.2. Data transfer

To activate the right portion of Figure 2.16, we call `H5Dwrite()`, as shown in Listing 2.11. The key elements of the user data journey are captured in Figure 2.10. We can see that storage for the data is allocated in the file. There is, however, a twist: the data is not written into the file when `H5Dwrite()` is called or completed. Because the array to be written is so small, the data is copied to a buffer, and the write is delayed until the dataset is closed, when it is flushed (`H5D__flush_sieve_buf()`). The native VOL uses a so-called data sieve buffer to reduce the number of small I/O operations. (We will have more to say about the sieve buffer in section 2.6.)

2.3.3. Chunked storage layouts, selections & partial I/O

The flow discussed in section 2.3.2 places us in the right portion of Figure 2.16. However, we haven't exercised much of the machinery shown there. To activate a few of those components, we modify the storage layout and write only parts of the dataset, as shown in Listing 2.12.

The changes compared to the sequence shown in Figure 2.10 are shown in Figure 2.11.

The gist is this:

1. The different storage layout (chunked) is noted in the dataset's object header at creation time. The chunk locations are tracked with a B-tree index. Not that, as in the contiguous case, chunk allocation is delayed until any actual data is written.
2. The hyperslab selection construction is a 100% in-memory metadata affair and is *not* routed through the VOL interface.

```c
#include "common.h"
int main() {
    hid_t file_id = H5Fcreate("data.1.h5", H5F_ACC_TRUNC, H5P_DEFAULTx2);
    hid_t fspace = H5Screate_simple(1, (hsize_t[]){10}, NULL);
    hid_t dset = H5Dcreate(file_id, "data", H5T_NATIVE_INT, fspace,
        H5P_DEFAULTx3);
    int data[10] = {0,1,2,3,4,5,6,7,8,9};
    H5Dwrite(dset, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT, data);
    H5Dclose(dset);
    H5Sclose(fspace);
    H5Fclose(file_id);
    return 0;
}
```

Listing 2.11 – Data transfer.

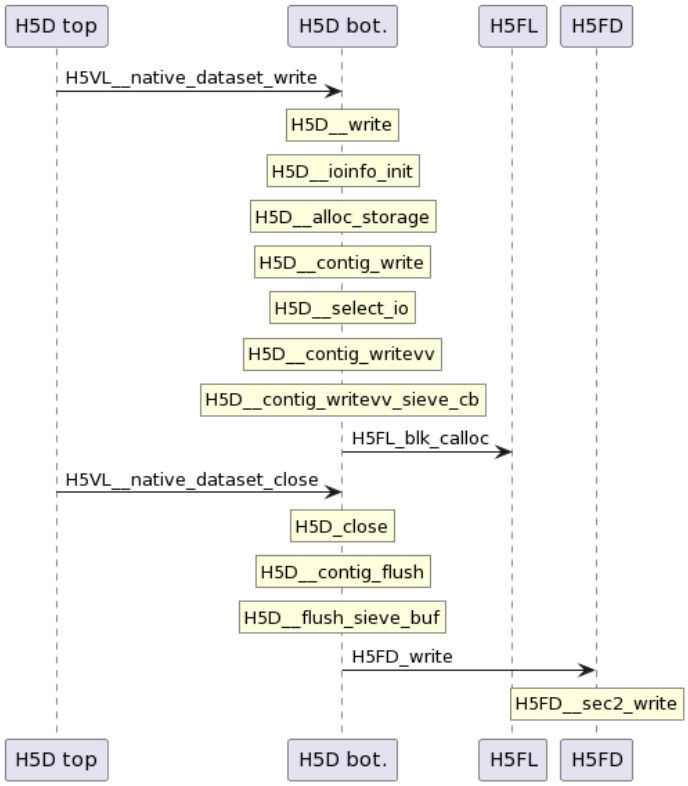


Figure 2.10. – Process to write a (small) contiguous dataset.

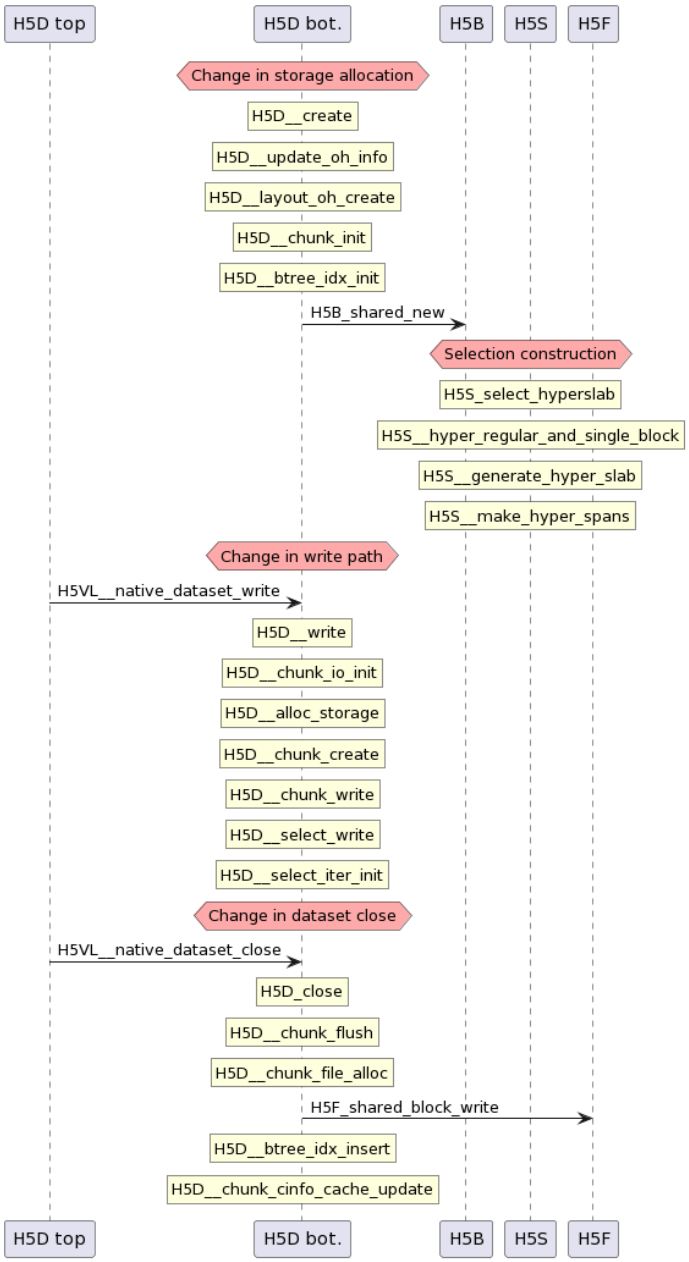


Figure 2.11. – Process to write a selection to a (small) chunked dataset.

```c
#include "common.h"
int main() {
    hid_t file = H5Fcreate("data.3.1.h5", H5F_ACC_TRUNC, H5P_DEFAULTx2);
    hid_t fspace = H5Screate_simple(1, (hsize_t[]){10}, NULL);
    hid_t dcpl = H5Pcreate(H5P_DATASET_CREATE);
    H5Pset_chunk(dcpl, 1, (hsize_t[]){6});
    hid_t dset = H5Dcreate(file, "data", H5T_NATIVE_INT, fspace,
        H5P_DEFAULT, dcpl, H5P_DEFAULT);
    int data[5] = {1,3,5,7,9};
    hid_t mspace = H5Screate_simple(1, (hsize_t[]){5}, NULL);
    H5Sselect_all(mspace);
    H5Sselect_hyperslab(fspace, H5S_SELECT_SET, (hsize_t[]){1}, (hsize_t[]){2},
        (hsize_t[]){5}, (hsize_t[]){1});
    H5Dwrite(dset, H5T_NATIVE_INT, mspace, fspace, H5P_DEFAULT, data);
    H5Dclose(dset);
    H5Sclose(mspace);
    H5Pclose(dcpl);
    H5Sclose(fspace);
    H5Fclose(file);
    return 0;
}
```

Listing 2.12 – Data – chunked layout, selection, & partial I/O.

3. The function `H5Dwrite()` is responsible for allocating chunks and iterating through selections. However, data is not written immediately if the array and chunk sizes are small. Instead, it is written only after the dataset is closed and flushed. The actual writing of user data is done by `H5F_shared_block_write()`. After writing, the chunk index is updated.

While we have not yet explored the native VOLs facilities for datatype conversion, data filtering, or data caching and buffering, we have seen enough to corroborate the claim that the native VOL does not alter user data. Instead, it frames the data and augments them with metadata to create a self-describing package called a native HDF5 file. In section 2.3.4, we explore how an everyday kind of user-metadata, attributes, fares under the scheme outlined in Figure 2.16.

2.3.4. User-defined metadata – The life cycle of an HDF5 attribute

Overview Attributes' purpose is to associate metadata with HDF5 objects directly. Attributes have a very similar interface to datasets - they have a layout defined by a dataspace, contain user data with a datatype determined at creation, and can be written to and read from. Attribute operations pass through the Virtual Object Layer to reach the active VOL connector's storage implementation.

Because attributes exist primarily as metadata, they are distinct from datasets in a few key ways. Attributes are always associated with a parent object: a dataset, group, or committed datatype. Attributes can only be accessed through their parent object, and attribute names are significant only within that parent object. Because they exist entirely as metadata in storage by default, attributes should generally contain less than 64KiB of data (see the "Attribute storage layouts" section for handling larger attributes). For the same reason,

```c
#include "common.h"
int main() {
  hid_t file_id = H5Fcreate("attrib.1.h5", H5F_ACC_TRUNC, H5P_DEFAULTx2);
  hid_t scalar = H5Screate(H5S_SCALAR);
  hid_t attr_id =
      H5Acreate(file_id, "meta", H5T_NATIVE_INT, scalar, H5P_DEFAULTx2);
  int meta = 42;
  H5Awrite(attr_id, H5T_NATIVE_INT, &meta);
  H5Aclose(attr_id);
  H5Sclose(scalar);
  H5Fclose(file_id);
  return 0;
}
```

Listing 2.13 – Attribute life cycle.

an attribute's data must be read or written entirely in every access; partial reading or writing using a selection is not allowed.

Attributes as objects Listing 2.13 shows the life cycle of an attribute. It is created on a parent object, in this case (the root group of) the file `attrib.1.h5`. It is created with a dataspace and datatype defining the data it stores. Data is written to the attribute in storage, and then the in-memory object for the attribute is closed.

When an attribute is opened, the library handle points to an instance of `H5A_t`. The shared attribute information attached to this structure contains pointers to the attribute's datatype and dataspace, as well as its creation index on its parent object if creation index tracking was enabled at object creation time. Because the creation index is only computed when the attribute is first created or opened, operations such as removing attributes could cause it to become invalid (e.g. its creation index could exceed the total number of attributes on the parent object). This is why, before an attribute is deleted from its parent object, identifiers of other attributes on that object should be closed, and only re-opened after the attribute deletion is complete.

2.3.5. Attribute storage layouts

Overview There are two primary ways that attributes may be stored - 'compact' storage and 'dense' storage. Compact attributes are small and few in number, and are stored in the header of their parent object. Dense attributes are large or many in number, and are stored in the file's global fractal heap. There is a technique that could be considered a third metadata storage method: Using attributes with reference datatypes to point at metadata stored in other objects.

Compact Storage When attributes are in compact storage, they are stored as attribute messages in the object header of their parent object. An attribute is stored compactly if it is below 64KiB in size and the parent object's total number of attributes is below its max compact storage threshold defined by `H5Pset_attr_phase_change()`.

```c
#include "common.h"
int main() {
  hid_t fapl_id = H5Pcreate(H5P_FILE_ACCESS);
  hid_t fcpl_id = H5Pcreate(H5P_FILE_CREATE);
  H5Pset_libver_bounds(fapl_id, H5F_LIBVER_V18, H5F_LIBVER_LATEST);
  /* Max in compact storage = 2, min in dense storage = 2 */
  H5Pset_attr_phase_change(fcpl_id, 2, 2);
  hid_t file_id =
    H5Fcreate("attrib.2.h5", H5F_ACC_TRUNC, H5P_DEFAULT, fapl_id);
  hid_t scalar = H5Screate(H5S_SCALAR);
  hid_t attr_id1 =
      H5Acreate(file_id, "attr1", H5T_NATIVE_INT, scalar, H5P_DEFAULTx2);
  hid_t attr_id2 =
      H5Acreate(file_id, "attr2", H5T_NATIVE_INT, scalar, H5P_DEFAULTx2);
  /* More than 2 compact attributes - all are moved to dense */
  hid_t attr_id3 =
      H5Acreate(file_id, "attr3", H5T_NATIVE_INT, scalar, H5P_DEFAULTx2);
  H5Aclose(attr_id1);
  H5Aclose(attr_id2);
  H5Aclose(attr_id3);
  H5Adelete(file_id, "attr2");
  H5Adelete(file_id, "attr3");
  /* Less than 2 dense attributes - the rest are moved to compact */
  H5Sclose(scalar);
  H5Fclose(file_id);
  H5Pclose(fapl_id);
  H5Pclose(fcpl_id);
  return 0;
}
```

Listing 2.14 – Dense storage used with many attributes

```c
#include "common.h"
int main() {
  hid_t fapl_id = H5Pcreate(H5P_FILE_ACCESS);
  H5Pset_libver_bounds(fapl_id, H5F_LIBVER_V18, H5F_LIBVER_LATEST);
  hid_t file_id =
    H5Fcreate("attrib.3.h5", H5F_ACC_TRUNC, H5P_DEFAULT, fapl_id);
  hid_t scalar = H5Screate(H5S_SCALAR);
  /* One element is 4*2*512*512 = ~4194K bytes (with 4-byte integers)
   * Will automatically be put into dense storage */
  hid_t dtype = H5Tarray_create(H5T_NATIVE_INT, 2, (hsize_t[]){512, 512});
  hid_t attr_id = H5Acreate(file_id, "image", dtype, scalar, H5P_DEFAULTx2);
  int meta[512][512];
  H5Awrite(attr_id, dtype, &meta);
  H5Aclose(attr_id);
  H5Tclose(dtype);
  H5Sclose(scalar);
  H5Fclose(file_id);
  H5Pclose(fapl_id);
  return 0;
}
```

Listing 2.15 – Dense storage used with a large attribute

Figure 2.12 is a diagram of the library's internal process to create an attribute in compact storage with the native VOL connector. First, `H5O__attr_create()` uses `H5O_pin()` to 'pin' the object header, preventing it from being flushed from the cache to storage until it is unpinned at the end of the operation. Once it is determined that the new attribute should be compact, `H5O__msg_append_real()` handles creating the new attribute message and inserting it into the object header. This is done in two steps: First, `H5O__msg_alloc()` creates a new message in the header, and returns its location within the header. Then, `H5O__copy_mesg()` uses the copy callback of the message type to populate the newly created message. In this case, the new message is an attribute message providing the copy callback `H5O__attr_copy()`, a wrapper around the library's function for copying attributes, `H5A__copy()`. The generic attribute copy function may be used here because attribute messages contain copies of the native `H5A_t` object.

Figure 2.13 is a diagram of the library's internal process to remove a compact attribute. Once again, the object header is pinned to prevent it from being flushed during an operation. The attribute messages in the object header are iterated over in `H5O__mesg_iterate_real()`, and the callback `H5O__attr_remove_cb()` is executed on each. If the attribute message matches the name of the attribute to be deleted, then the deletion process is performed by `H5O__release_mesg()`. The removal of the object header message takes place in two parts: first, `H5O__delete_msg()` is used to invoke the attribute message's delete callback, `H5O__attr_delete()`, which deletes header messages that were referred to by this attribute - specifically, dataspace and datatype messages. After that, `H5O__msg_free_mesg()` invokes the attribute message's 'free' callback, to free the native information stored in the attribute message - the `H5A_t` structure and its associated memory.

Compact attributes have some drawbacks - performance suffers when objects have many compact attributes and they are strictly limited in size to fit within the object header. Both of these limitations are circumvented by dense storage.

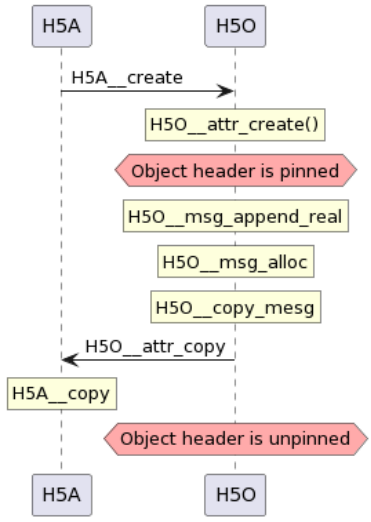


Figure 2.12. – Process to create a compact attribute

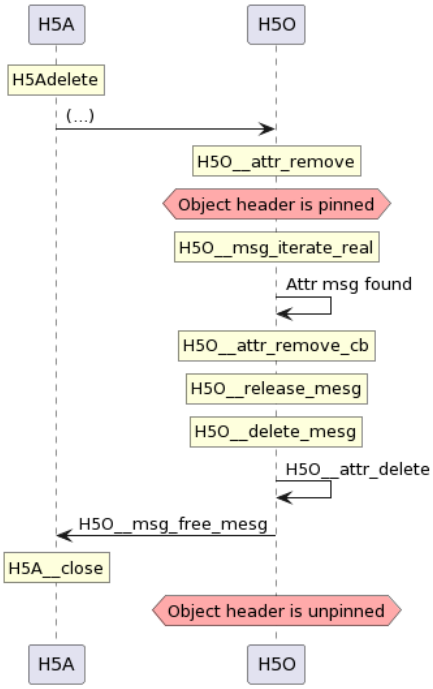


Figure 2.13. – Removal of a compact attribute

Dense Storage When an attribute is in dense storage, the object header contains a pointer to the location of the attribute's data on disk rather than containing the attribute's data directly.

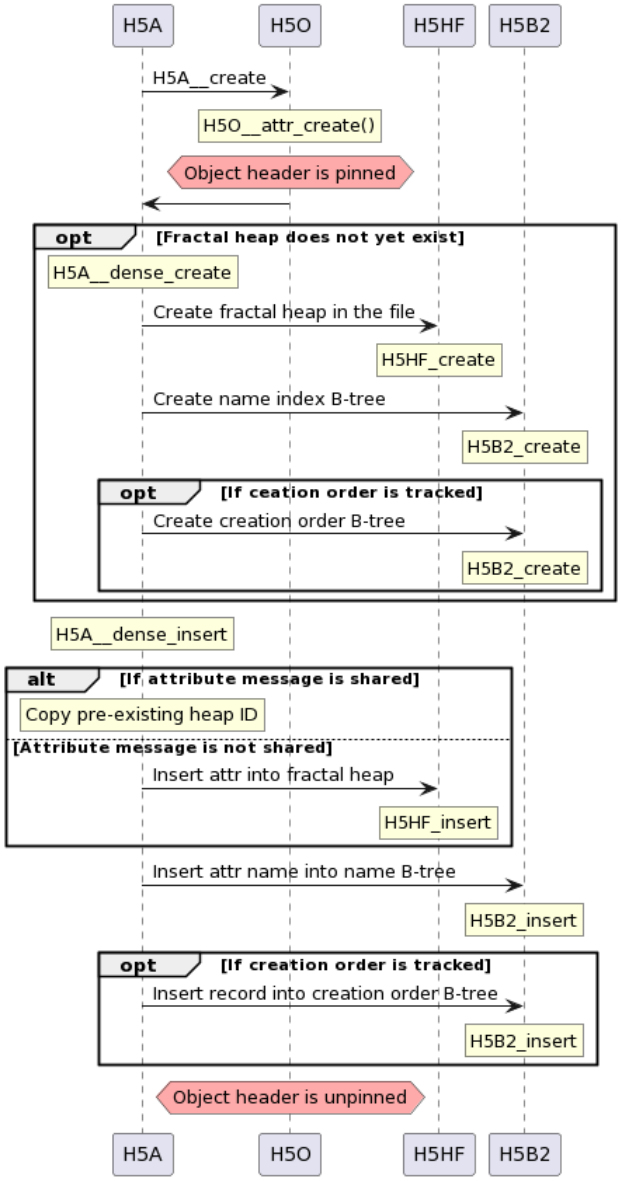


Figure 2.14. – Attribute insertion into dense storage

There are two circumstances under which a new attribute is placed into dense storage. The first is that the number of attributes on the parent object exceeds the maximum number of compact attributes on the object. This threshold may be specified for each object using `H5P_set_attr_phase_change()`. The second is that the size of the attribute exceeds 64k bytes, and it would not fit into compact storage. All pre-existing attributes are moved to dense storage whenever a new attribute is placed into dense storage.

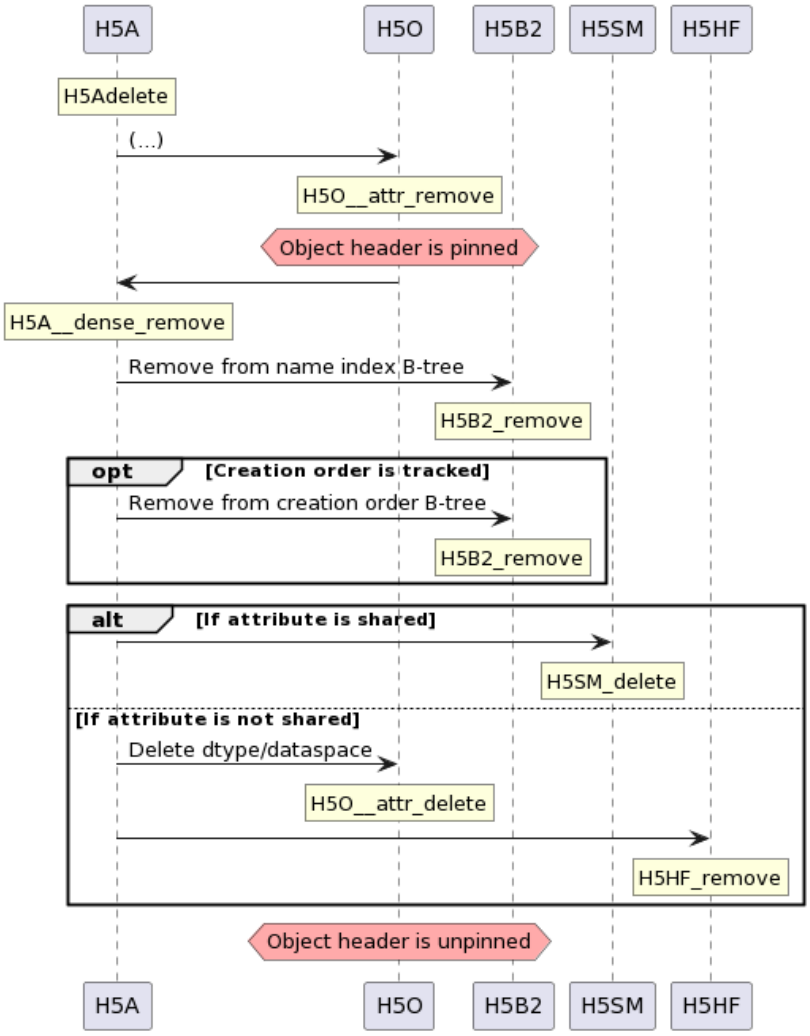


Figure 2.15. – Attribute deletion in dense storage

Figure 2.14 shows the library's internal process to create an attribute in dense storage. The object header is still pinned, even though the new attribute is not to be stored in the object header. This is because the attribute creation may have side effects that require modifying the object header. If this file does not yet have a fractal heap for dense attribute storage, one is created with `H5A__dense_create()`. This step also involves the creation of a B-tree to map dense attribute names to attribute information and optionally creating a B-tree to make the attribute information available by creation order. Once the dense attribute storage structures are confirmed, the attribute's information is stored densely by `H5A__dense_insert()`. This involves using the attribute message's encode callback to serialize the attribute, and this serialized attribute information is inserted into the fractal heap by `H5HF_insert()`. If the attribute is shared, then this attribute points to the information of an attribute that already exists in the fractal heap. In this case, the insertion into the fractal heap is skipped, and the heap ID of the existing attribute in the fractal heap is copied. The heap information of the new attribute (or preexisting shared attribute) is stored in the name-index B-tree, to be accessed later by the hash of the attribute name. Lastly, if the creation order is tracked, the heap information is also inserted into the creation-order B-tree, to be accessed later by link creation order.

Figure 2.15 shows the library's internal process to remove an attribute from dense storage. After `H5O__attr_remove()` determines the target attribute is in dense storage, `H5A__dense_remove()` performs the removal. `H5B2_remove()` searches the name-index B-tree and removes the information for the dense link. If creation order is tracked, the same operation is performed on the creation order B-tree. If the dense attribute is shared between objects, the 'deletion' is handled by using `H5SM_delete()` to decrease the reference count of the shared attribute message. Otherwise, the dense attribute's referenced information (dataspace/datatype messages) is deleted by `H5O__attr_delete()`, and it is removed from the dense attribute fractal heap.

When an attribute is removed from storage, the remaining attributes may be moved back into compact storage. If two conditions are met, an attribute will be moved back to compact storage. First, the total number of remaining attributes must be below the minimum number of dense attributes on the object, another threshold set by `H5P_set_attr_phase_change()`. Second, the attribute must be under 64k bytes in size. It is possible for the removal of an attribute to cause some, but not all, of the remaining attributes to be moved to compact storage due to some being too large to store compactly.

Attribute Storage and File Format Versions Note that because it affects how the HDF5 file is stored on disk, using dense attribute storage requires allowing the library to use a 1.8+ version of the file format (i.e. using `H5Pset_libver_bounds()` with a low version $\geq$ 1.8). With a pre-1.8 compatible file format version, attribute storage will always be compact, and attempts to create attributes greater than 64KiB in size will fail. Pre-1.8 compatible large metadata storage can be accomplished by storing metadata as datasets.

2.3.6. Summary

Attributes provide a flexible way to store metadata on an HDF5 object with the same CRUD interface as datasets. By default, attributes are stored compactly in the object header with other object metadata. The library takes advantage of the properties of B-trees and fractal heaps for large attributes or large amounts of attributes. If backward compatibility or compressibility is required for metadata, it may be stored in datasets and accessed from other objects through references.

From the perspective of the native VOL, user metadata and data travel down two different “swim lanes," as depicted in Figure 2.16. The main reason is their different role and functions in the HDF5 data model, which leads to different operations being supported in efficient and practical implementations.

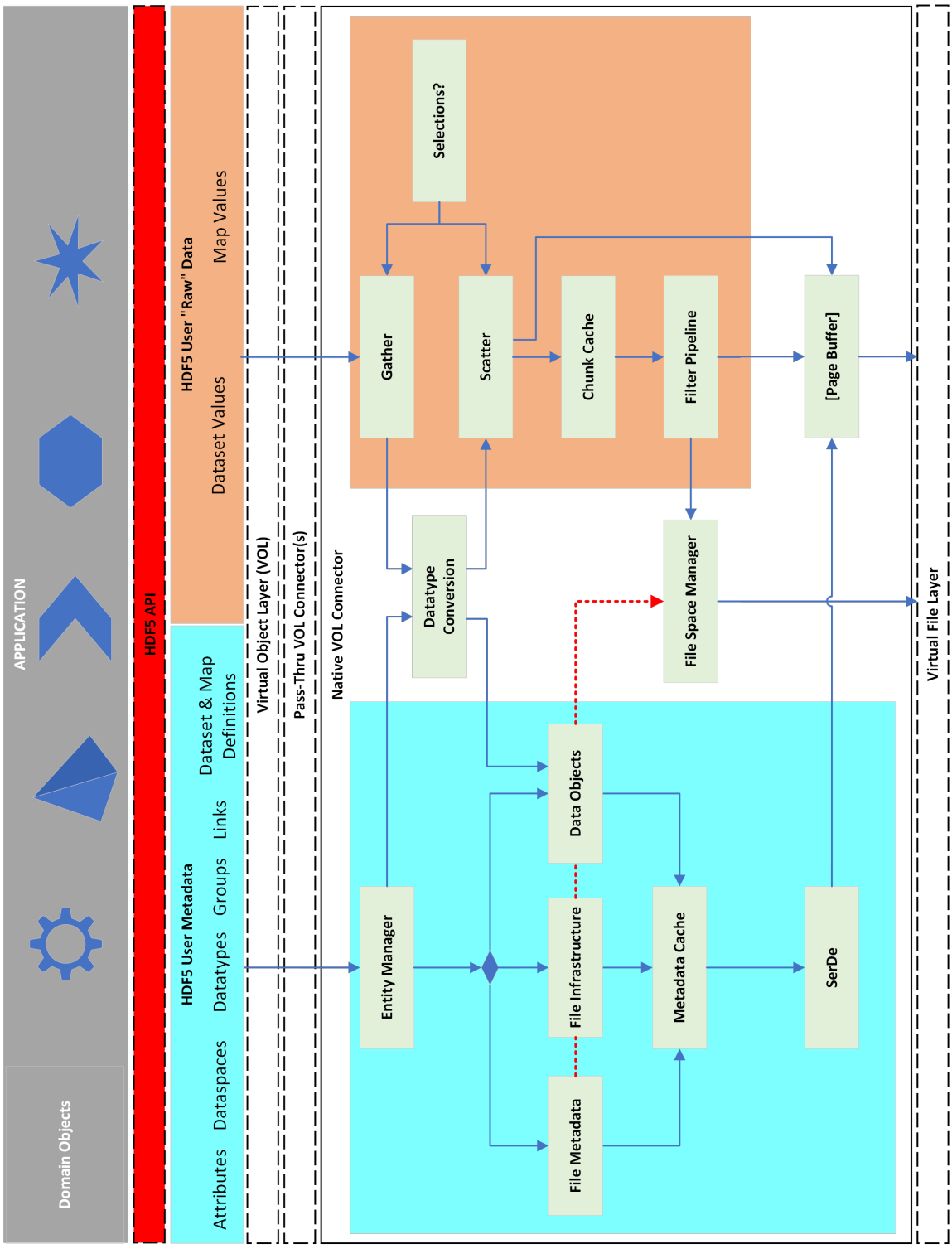


Figure 2.16. – The different I/O paths for metadata and "raw" data through the HDF5 library.

```c
#include "common.h"
int main() {
  hid_t file_id = H5Fcreate("groups.1.h5", H5F_ACC_TRUNC, H5P_DEFAULTx2);
  hid_t group_id = H5Gcreate(file_id, "grp1", H5P_DEFAULTx3);
  H5Gclose(group_id);
  H5Fclose(file_id);
  return 0;
}
```

Listing 2.16 – Group life cycle.

2.4. The H Stands for Hierarchical

A somewhat twisted reading of 'Hierarchical Data Format' might create the idea that there might be something hierarchical about the file format. There indeed are a few discernible layers in the file format specification [10], and we used the TCP/IP protocol metaphor (see Figure 2.8) to explain how the native VOL frames user metadata and data into portable self-describing packages, but this would be a rather uninteresting hierarchy, and none that would help users. What's hierarchical about HDF5 is how the object namespace is realized through groups and links, which differs fundamentally from hierarchies that can be mimicked in key-value representations. Not only is the file-system-in-a-file metaphor apt, but an HDF5 file is also a *web-in-a-file*.

2.4.1. Groups

Overview Groups are a key aspect of an HDF5 file's hierarchical structure. A minimal example of group creation is shown in Listing 2.16.

It is often said that named HDF5 objects (datasets, groups, and committed datatypes) reside in groups. To be more precise, groups contain link objects that point to named HDF5 objects. When a named object is created, a link that points to it is also created. It is this link that is written to the group's object header in storage. This is why `H5Dcreate()`, `H5Tcommit()`, and `H5Gcreate()` all take a link creation property list as an argument.

Figure 2.17 shows the library's internal process to create a new group through the native VOL. The application calls the API function `H5Gcreate()`, and because groups are (generally) named objects, `H5G__create_named()` is invoked. After creating a new group in an HDF5 file, `H5L_link_object()` links it into the HDF5 file hierarchy. This linking process involves creating a link using `H5L__create_real()` that points to the newly created group. The link is then stored in the parent group of the new object, which, in this case, is the root group.

The Root Group Every HDF5 file is created with a single named object - the root group. The root group has the fixed name `/`, and acts as the root of a file's hierarchy. It is the only group that does not need to reside within a group itself. The root group is closely associated with the file itself - when API functions operate on HDF5 files as objects, they often act on the root group internally.

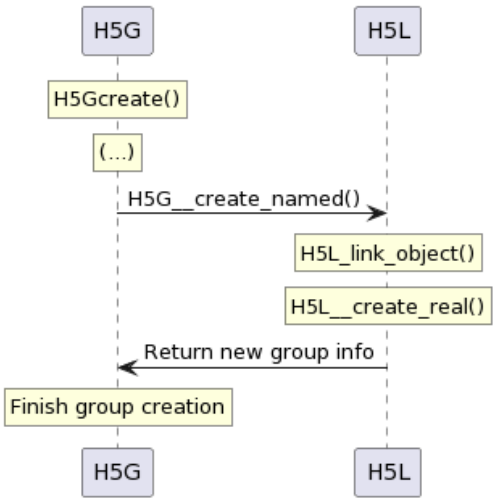


Figure 2.17. – The library internally creates a link to a new named group

2.4.2. Links

Overview The structure of an HDF5 file is a rooted, directed graph. Named objects form the nodes of this graph, and links are the edges between them. Links point from a group to a target object, and may be 'traversed' from their parent group to access the target object. Links are indexed within a group, allowing them to be iterated through recursively with `H5Lvisit()` or non-recursively with `H5Literate()`.

When a named object is created in a particular group, a hard link pointing to it is created and placed in that group. Hard links store the physical address of the object in the file. Other types of links, which use a name or other symbol to point to an object rather than its physical address, are called 'symbolic links'.

Hard Links and Objects Each time a hard link to an object is created or destroyed, the target object's internal count of how many links point to it, the field `nlink` in the `H5O_t` struct, is updated accordingly.

When an object has no links pointing to it, it becomes inaccessible in the file and the space it occupies is marked as 'free' by the file's free space manager. Unless persistent free space management is enabled on the file, this free space tracking information will be lost when the file is closed, and the memory will not be reclaimed until a tool such as `h5repack` is used.

In the context of HDF5, "reference count" refers to in-memory objects tracking how many other in-memory objects reference them, and the term has no relation to the number of links to an object in the file hierarchy.

Link Creation Figure 2.18 illustrates the library's process for link creation. After link-type-specific setup is done by `H5L__create_[hard/soft/external/ud]()`, the link object is created by `H5L__create_real()`.

The API's link creation functions require both a location identifier and a name for the link that will be created. This link name may be a relative path from the object specified by the location identifier, or an absolute path from the root of the file. If the provided link name is a path with intermediate elements, then

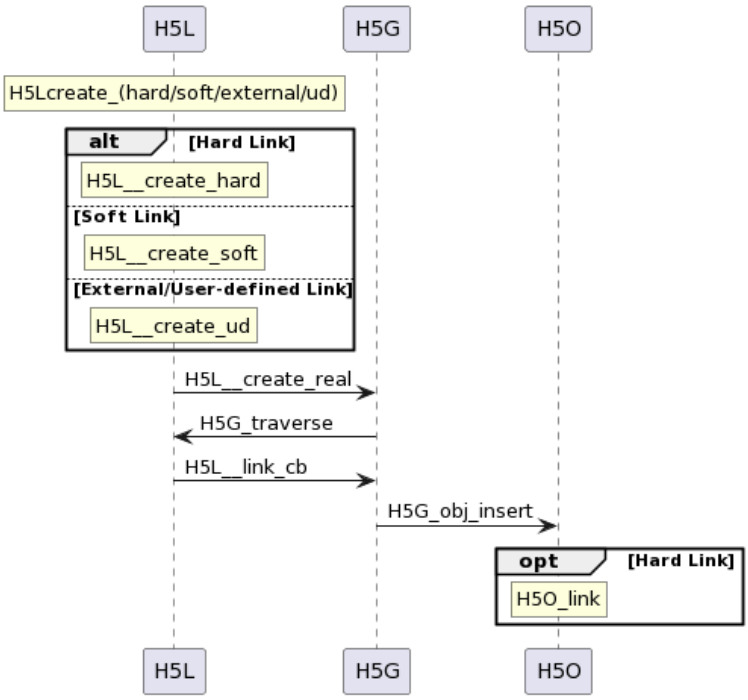


Figure 2.18. – Link creation sequence diagram

`H5G_traverse()` is used to traverse the path and create the link object at the correct place in the file hierarchy. Once the final destination for the new link is reached, the link is inserted into the group containing it with `H5G_obj_insert()`. If it is a hard link, `H5O_link()` increments the target object's `nlink` count.

Soft Links Soft links are a type of symbolic link that point to a path in the file hierarchy rather than an address in the file. When a soft link is created, no checks are performed to ensure that an object exists at the target path. A link that does not point to an existing object is called a dangling link. Allowing soft links to dangle means that changing, removing or entirely replacing an object that resides at a location pointed to by soft links can be done without additional work on the part of the library or the application.

Accessing Objects Through Links Figure 2.19 shows the process the library uses to access an object through a link.

After the VOL layer is passed through and the native `H5D__open_name()` is invoked, `H5G_loc_find()` is called in order to retrieve the two components of the location of the dataset - the path to it in the file hierarchy and its address in physical storage. This function invokes `H5G_traverse()` to traverse the path to the link, as well as the path the link contains if it is a soft or external link. `H5G_loc_find()` also provides the traversal function with the `H5G__loc_find_cb()` callback to be used upon locating the target object. This callback copies the target object's group location information. After the location information is retrieved, the rest of the type-specific object opening work is performed.

```c

#include "common.h"
int main() {
  hid_t file_id = H5Fcreate("groups.1.h5", H5F_ACC_TRUNC, H5P_DEFAULTx2);
  hid_t group_id = H5Gcreate(file_id, "grp1", H5P_DEFAULT, H5P_DEFAULTx2);
  hid_t space_id = H5Screate(H5S_SCALAR);

  H5Lcreate_soft("/grp1/data", group_id, "link_to_data", H5P_DEFAULTx2);
  hid_t dset_id = H5Dcreate(group_id, "data", H5T_NATIVE_INT, space_id,
    H5P_DEFAULTx3);
  H5Dclose(dset_id);

  dset_id = H5Dopen(group_id, "link_to_data", H5P_DEFAULT);
  H5Dclose(dset_id);
  H5Sclose(space_id);
  H5Gclose(group_id);
  H5Fclose(file_id);
  return 0;
}

```

Listing 2.17 – Soft link example

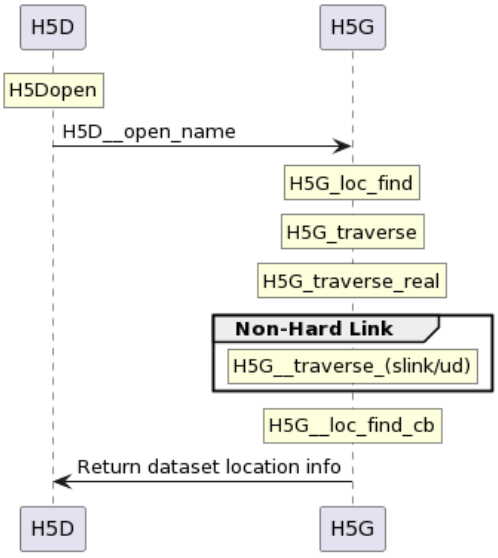


Figure 2.19. – Sequence diagram of opening a dataset through a link

```c
#include <string.h>
#include "common.h"
hid_t UD_soft_traverse(const char *link_name, hid_t cur_group,
                       const void *udata, size_t udata_size,
                       hid_t lapl_id, hid_t dxpl_id) {
  const char *target_obj_name = (const char *)udata;
  hid_t ret_value = H5Oopen(cur_group, target_obj_name, lapl_id);
  return ret_value;
}

const H5L_class_t UD_soft_class[1] = {{
    H5L_LINK_CLASS_T_VERS,              /* Version number for this struct */
    (H5L_type_t)H5L_TYPE_EXTERNAL + 1, /* Link class id number.          */
    "UD_soft_link",                     /* Link class name for debugging  */
    NULL,              NULL, NULL,      /* Link class op callbacks        */
    UD_soft_traverse, NULL, NULL
}};

int main() {
  hid_t file_id = H5Fcreate("ud_link.h5", H5F_ACC_TRUNC, H5P_DEFAULTx2);
  H5Lregister(UD_soft_class);
  /* Pass the path to the root group as udata */
  H5Lcreate_ud(file_id, "ud_link", (H5L_type_t)H5L_TYPE_EXTERNAL + 1, "/",
               strlen("/") + 1, H5P_DEFAULTx2);

  hid_t group_id = H5Gopen2(file_id, "ud_link", H5P_DEFAULT);
  H5Gclose(group_id);
  H5Fclose(file_id);
  return 0;
}

```

Listing 2.18 – Creating and using a user-defined link class

User-defined Links As part of the library's effort to be as extensible as possible, users may define and register their own link types. `H5Lregister()` can be used to register a new link class, after which an instance of the new link class may be created with `H5Lcreate_ud()`.

Listing 2.18 shows a minimal example of a user-defined link class, which mimics the behavior of soft links. The only callback that must be defined is the traversal callback, which handles obtaining an object handle from a link pointing to that object. The other callbacks - create, move, copy, delete, and query - all have some baseline functionality implemented by the library, with the user's callback being invoked after the internal work is completed. For example, for a user-defined link class with no provided query callback, `H5Lget_info()` will still return the character set of the link's name, its creation order in its parent group, whether its creation order is valid, and the link's type. However, the returned size of the link will default to zero unless the query operation is implemented.

2.4.3. Link Storage

Overview Similar to attributes, there are two ways that links may be stored in a file: compact and dense. Compactly stored links are kept in the object header for the group containing them, along with other metadata for that group object. Densely stored links are kept in a fractal heap. The address of the dense link fractal heap is stored in the Link Info message in the group's header, with the heap IDs to access specific dense links stored in a name index B-tree.

Older versions of the file format stored all links in a symbol table. A symbol table will still be used for link storage if the 'low' library compatibility version on a file is set lower than 1.8. By default, a new file has its 'low' compatibility version set to `H5F_LIBVER_EARLIEST`, which will disable use of the fractal heap for dense link storage.

Links are stored compactly if two criteria are met: the link message is under 64 KiB in size, and the number of links is below the maximum number of compact links on the group. The maximum number of compact links on a group may be set at creation time via `H5Pset_link_phase_change()`. If either of these criteria are not met, the new link will be put into dense storage, and any existing compact links will be converted to dense storage. If the number of links later drops below the minimum number of compact links set by `H5Pset_link_phase_change()`, then any links below 64 KiB will be converted to compact storage.

Inserting Links into Storage An example of link insertion into dense storage is illustrated in Figure 2.20. If the file does not yet have the structures needed to store dense links, they are created in `H5G__dense_create()`. This involves creating a fractal heap for dense links via `H5HF_create()`, creating a B-tree to hold the names of dense links via `H5B2_create()`, and, if creation order is tracked on the parent group, creating another B-tree to track creation order. Once the the structures for dense storage are confirmed to exist, insertion into dense storage is handled by `H5G_dense_insert()`. The insertion into the fractal heap itself is done by `H5HF_insert()`. The name-index B-tree stores information about the link that may later be accessed by the hash of the link name, and the creation-order B-tree stores the same information with the link creation index as the key.

The process for compact link insertion into the object header is illustrated in Figure 2.21. Compact links are inserted by `H5G__compact_insert()`. `H5G_msg_create()` sets up the message to be inserted. Before the message is appended onto the object header, the object header is "pinned" with `H5O_pin()`. Pinning the object header prevents it from being written from cache to storage in the middle of operations

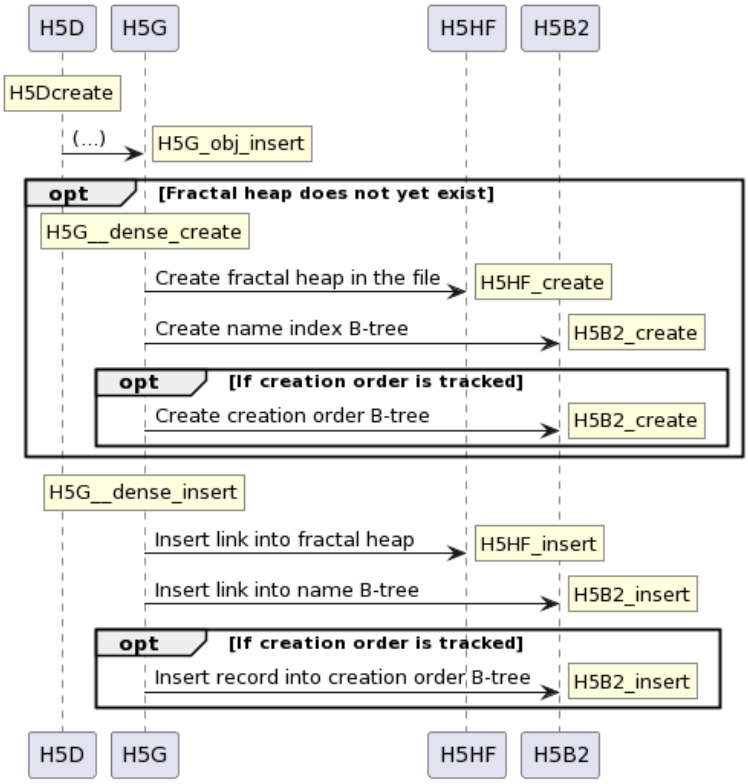


Figure 2.20. – Link insertion into dense storage during dataset creation

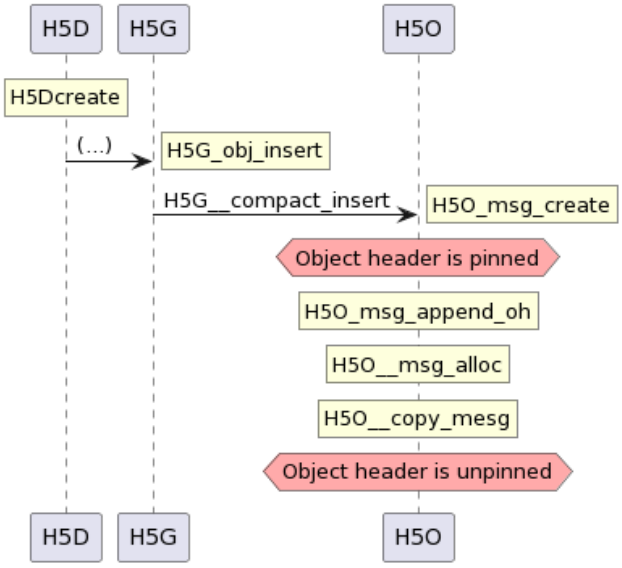


Figure 2.21. – Link insertion into compact storage during dataset creation

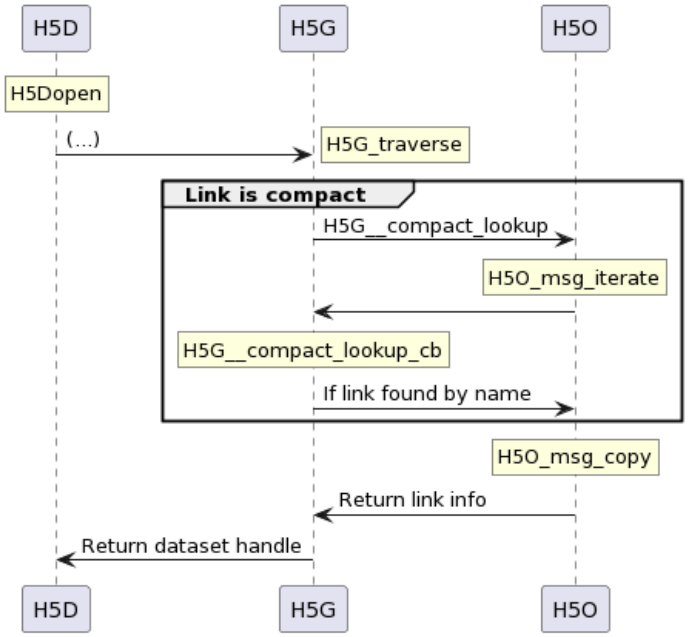


Figure 2.22. – How the library accesses a compact link

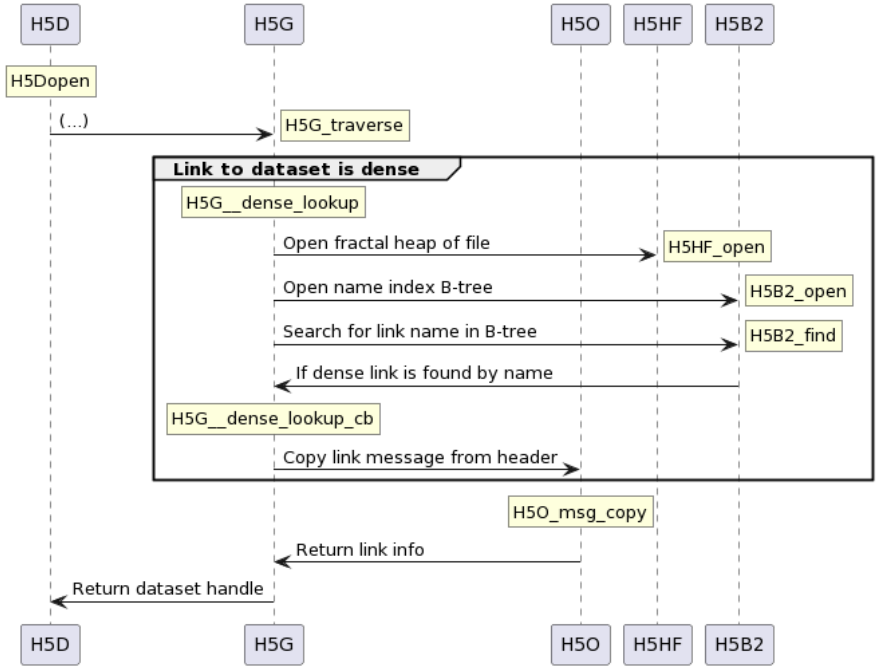


Figure 2.23. – How the library accesses a dense link

that modify the header. `H5O_msg_append_oh()` appends the newly created message to the object header. This is done in two parts: first `H5O__msg_alloc()` creates space in the object header, then `H5O__copy_mesg()` populates the allocated memory with link message information. Lastly, the object header is unpinned by `H5O_unpin()`, allowing it to be flushed from cache again.

Accessing Links in Storage Figures 2.22 and 2.23 illustrate the library's process for accessing objects through links in compact and dense storage, respectively.

`H5G_traverse()` is used regardless of storage type to locate the link through the provided path in the file hierarchy. Once the correct location in the hierarchy is reached, the process branches based on how the link is stored.

If the link is compact, `H5G__compact_lookup()` searches the object header for a message describing a link of the correct name. `H5O_msg_iterate()` iterates through each message and uses `H5G_compact_lookup_cb()` to copy the link information if a match is found.

If the link is dense, `H5G__dense_lookup()` does the name lookup. The dense link fractal heap and the name index B-tree are both opened. `H5B2_find()` searches name index B-tree for the dense link. If a match is found, the provided callback `H5G__dense_lookup_cb()` copies the link information.

In both cases, once the link is found by name, its information is copied with `H5O_msg_copy()` and returned to `H5G_traverse()`. The link traversal callback for the link's type is used to access information about the object through the link, and a handle to the object is returned by the top-level open function.

2.4.4. Summary

Implementing group membership via links introduces a layer of indirection to the file hierarchy which is extremely powerful. Based on the use case of the application, objects can be accessed based on storage location, path within the file hierarchy, path in the filesystem via external links, or something else entirely with user-defined link classes. Since scalability with extremely large amounts of data is a core goal of the HDF5 library, maximizing the user's ability to manipulate the file hierarchy without affecting the underlying arrangement of data is critical.

Scalability with large numbers of links in a group was another important design goal, realized in the architecture by dense link storage. The tradeoff between compact storage being better for small numbers of links and dense storage being better for large numbers of links is automatically handled during operation by the maximum compact/minimum dense thresholds, and may also be custom-fit to a use case by the user through manipulating property lists.

2.5. Stoked on datatypes

Continuing in the educational tours, we will delve into the intricate details of datatypes, their categories, and how they are represented and managed by the HDF5 library. We will also explore the encoding process of datatypes for disk storage and how the library handles data of different data types in the file.

Moreover, we will also highlight the remarkable extensibility and customizability of the HDF5 library, which empowers users to define their own datatypes, composite datatypes, and datatype conversion functions. This capability allows for optimizing almost every I/O process step.

2.5.1. Datatype life cycle

Speaking broadly, the datatype life cycle begins at build configuration time with the discovery of platform-specific types, proceeds at run-time during library initialization, and continues with the construction of predefined and user-defined datatypes. In this section, we describe the library infrastructure supporting this process.

Configuration-Time Type Discovery The HDF5 library checks the size of each built-in C type on the current platform at configuration time. A set of macros with names of the form `H5_SIZEOF_<TYPE>` will be defined in the generated file `H5pubconf.h`. These macros define an internal assignment operator between numeric types that works regardless of the types' relative sizes, `H5_CHECKED_ASSIGN`.

Predefined Datatypes The types predefined by the library have standard symbolic names of the form `H5T_<arch>_<base>` where 'arch' is an architecture name, and 'base' is a programming type name formed from a letter for the type class, a number for the precision in bits, and an indication of byte order. For example, `H5T_INTEL_I64BE` is a predefined type for 64-bit big-endian integers on Intel CPUs.

The names through which the predefined types are accessed are defined as macros in `H5Tpublic.h`. The macro `H5T_<TYPE>` evaluates to `H5OPEN H5T_<TYPE>_g`. `H5OPEN` is a macro that ensures the library is initialized if the type macro was used in a public or user application file. `H5T_<TYPE>_g` is a globally accessible handle ID for the predefined type.

Predefined types' inability to be modified is implemented through the `state` field of `H5T_t`'s shared information. Predefined types are created with their state set to immutable, making them unmodifiable and unable to be closed by the user.

Predefined Datatype Initialization The H5T package is initialized during phase 2 of H5VL's initialization, as seen in Listing 2.3. An internal diagram of `H5T_init()` is shown in Figure 2.24.

Predefined types that match the native platform are created first in two steps. In the first step, predefined float datatypes that match the native platform are created and registered in `H5T__init_native_float_types()`. The `DETECT_F` macro detects the native platform's floating-point properties, which are used to construct matching datatypes. The datatypes are assigned library handles by `H5I_register()`. These handles are stored in a set of global variables with names of the form `H5T_NATIVE_<TYPE>_g`. In the second step, a similar process is carried out for all other native datatypes in `H5T__init_native_internal()`, using detected information about the platform's byte order and alignment.

Once the predefined native datatypes are created, they are used to create the predefined non-native datatypes through the macro `H5T_INIT_TYPE`. If the new type to be constructed is based on an existing one, this macro copies the provided base type with `H5T_copy()` before making size and byte order modifications. If the new type is not based on an existing type, the macro directly allocates space for a new `H5T_t`

structure and populates it using another macro of the form H5T_INIT_TYPE_<TYPE>_CORE. In both cases, H5T_INIT_TYPE registers the newly created datatype with the library, assigns it a handle through H5I_register(), and makes the handle globally accessible through H5T_<ARCH>_<BASE>_g. A user can then access these handles through macros of the form H5T_<ARCH>_<BASE>, as discussed in the previous section.

In addition to initializing predefined datatypes, H5T_init() also sets up many internal datatype conversion routines through H5T__register_int(), and registers the default property list for datatype creation, H5P_DATATYPE_CREATE, through H5P_create_id().

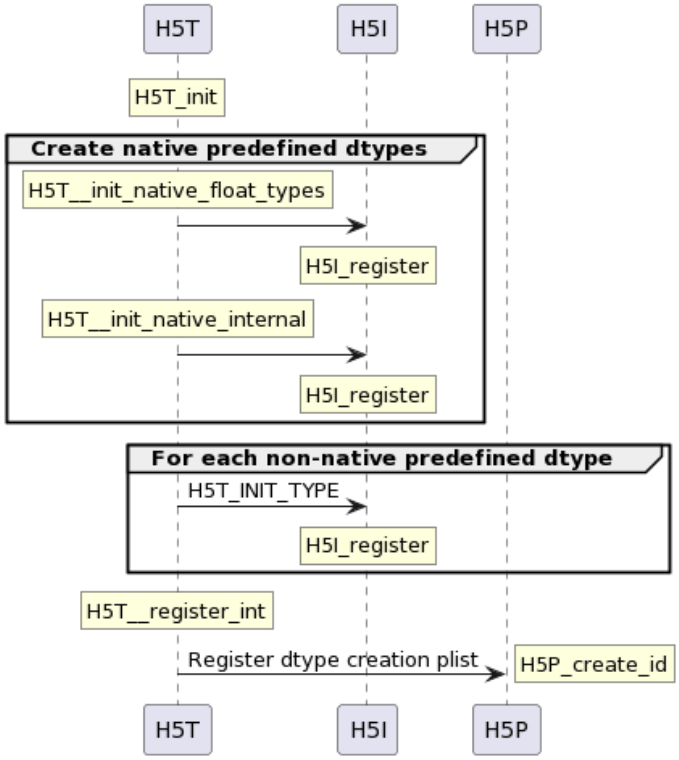


Figure 2.24. – A sequence diagram for H5T_init

User-defined Datatypes Users may derive their own types from the predefined types by copying them to get a transient, modifiable copy. Figure 2.25 is a minimal example demonstrating user-defined type creation. Figure 2.26 shows the internal operation of H5Tcopy().

Figure 2.25. – Creation of a user-defined type identical to a platform-native integer.

```
int main(void) {
    hid_t type_id = H5Tcopy(H5T_NATIVE_INT);
    H5Tclose(type_id);
};

```

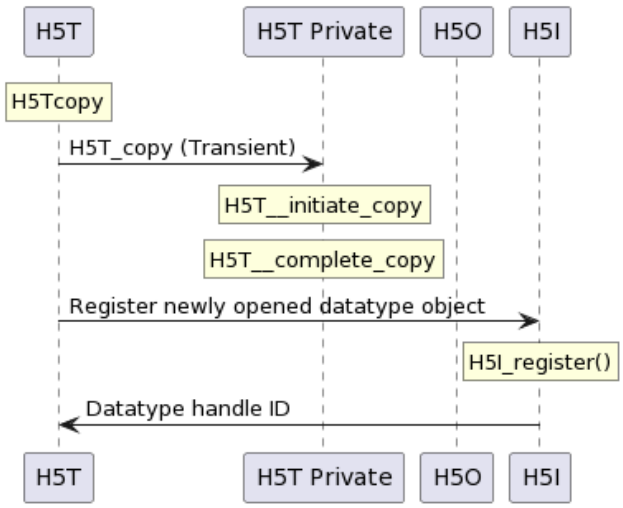


Figure 2.26. – H5Tcopy sequence diagram

```
int main(void) {
    hid_t file_id = H5Fcreate("type_test.h5", H5F_ACC_TRUNC, H5P_DEFAULTx2);
    hid_t type_id = H5Tcopy(H5T_NATIVE_INT);

    H5Tcommit(file_id, "user_defined_integer", type_id, H5P_DEFAULTx3);

    H5Tclose(type_id);
    H5Fclose(file_id);
};

```

Listing 2.19 – Committing a type as an HDF5 Object in storage

`H5Tcopy` performs the datatype copy operation in two steps. First, `H5T__initiate_copy()` allocates in-memory structures for the new object and, if the original datatype is committed, increases its in-memory reference count. Then `H5T__complete_copy()` copies the information for the new datatype from the old datatype. Lastly, `H5I_register()` assigns a handle ID to the new datatype.

Committed Datatypes There are two kinds of datatypes: transient datatypes that exist only in memory and committed (formerly "named") datatype objects that are written to storage.

Listing 2.19 shows a minimal program that turns a transient datatype into a committed datatype via `H5Tcommit()`. After data is written to the file, the datatype will exist as an object on the file in storage.

Figure 2.27 shows the internal work done by `H5Tcommit()` when using the native VOL. A copy of the provided datatype is created by `H5T_copy()`, and this copy is written to storage. `H5T__commit_named()` sets up creation information for `H5L_link_object()`, which creates the committed datatype object and a hard link used to link it into the file hierarchy. Lastly, the newly committed datatype object

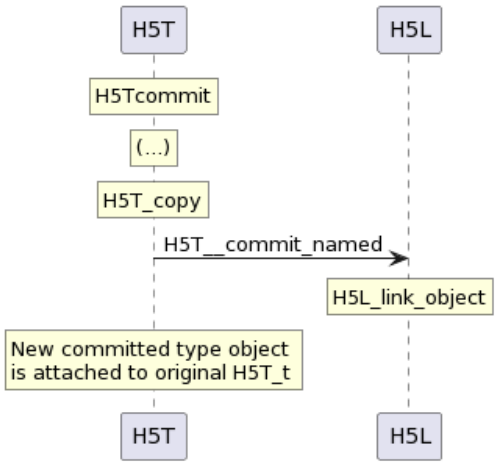


Figure 2.27. – Internal operation of H5Tcommit

is attached as an `H5VL_object_t` instance to the `vol_obj` field of the original datatype provided to `H5Tcommit()`.

2.5.2. Datatype conversion

In this section, we describe the circumstances in which and where datatype conversion takes place during user data transfer and how the library locates appropriate datatype conversion functions at run-time.

Overview Datatype conversion is performed at transfer (read or write) time. Which conversions are possible depends on the set of defined conversion functions. As a rule of thumb, conversion is possible if the datatypes of the source and destination are different types within the same datatype class. For example, integers can generally be converted to other integers and floats to other floats. Some other common conversions, like integer-to-float and vice versa, are also predefined.

Conversion Functions Conversion between a source and destination type is performed via a conversion function. A conversion function may be 'hard' or 'soft'. Hard conversion functions are each associated with a particular source and destination datatype, while soft conversion functions are related to entire datatype classes.

The library maintains a global table of type conversion functions, `H5T_g`. Each conversion function is uniquely identified by a "path" (`H5T_path_t`) formed from the source and destination datatypes. When a conversion is necessary, a binary search is performed on this table to locate an appropriate conversion function.

User-Defined Conversion Functions Users can add new conversion functions or overwrite the behavior of existing conversion functions through `H5Tregister()`. A new hard conversion function will overwrite any previous conversion functions for the specified types, and a new soft conversion function will overwrite

the conversion functions for any types to which it is applicable that do not already have a hard conversion function.

Listing 2.20 shows a user-defined conversion function `convert()`, which converts between the user-defined compound datatypes `src_t` and `dst_t`. Notice that the conversion operation, `H5T_CONV_CONV`, occurs entirely within the main buffer. This is straightforward in this case because `src_t` is larger than `dst_t`, and so the assignment to `((dst_t*) buf)[i].b` will never overwrite any memory needing to be read to convert a subsequent element. More complex conversions may need to use the 'background buffer', `bkg`, as a place for temporary storage.

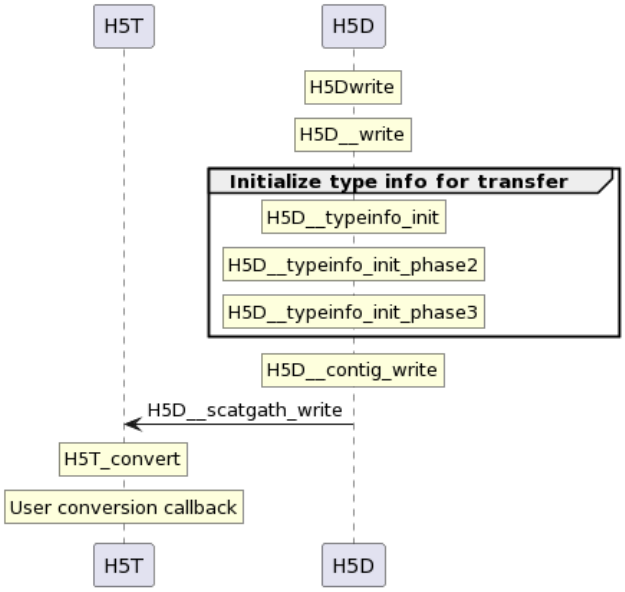


Figure 2.28. – Datatype conversion with user callback diagram

Listing 2.21 creates compound datatypes corresponding to `src_t` and `dst_t`, and registers `convert()` as a soft conversion function from`src` to `dst`. It invokes the library's top-level conversion function `H5Tconvert()`, automatically finding the newly registered conversion function by a path.

Datatype conversion can also be done automatically by the library during writes or reads. Listing 2.22 shows an example of a program where a user-defined conversion between floats is done automatically during a write to a dataset.

Figure 2.28 illustrates how type conversion happens during a write to a dataset. The library's internal dataset write function `H5D__write()` initializes datatype information in three phases. The initialization is done in phases due to requirements imposed by parallel I/O. Specifically, when doing parallel I/O, the maximum datatype size across all destination objects must be computed after the first phase, and the selection in the destination datasets must be adjusted after the second phase.

The first phase of type info initialization, `H5D_typeinfo_init()`, populates the `type_info` field of each dataset's `H5D_dset_io_info_t` structure with the appropriate datatype conversion function, the size of each datatype, and whether this is the special case of converting between compound datatypes that are subsets of one another. Unlike the other phases, the first phase is done separately for each destination dataset.

```c
#include "common.h"

typedef struct src_t {
     uint32_t a;
     float b;
  } src_t;

typedef struct dst_t {
     float b;
  } dst_t;

herr_t convert(hid_t src_id, hid_t dst_id, H5T_cdata_t *cdata,
    size_t nelmts, size_t buf_stride, size_t bkg_stride, void *buf,
    void *bkg, hid_t dxpl)
{
  herr_t retval = EXIT_SUCCESS;
  switch (cdata->command)
  {
  case H5T_CONV_INIT:
    printf("Initializing conversion function...\n");
    break;
  case H5T_CONV_CONV:
    printf("Converting...\n");
    for (size_t i = 0; i < nelmts; ++i)
      ((dst_t*) buf)[i].b = ((src_t*) buf)[i].b;
    break;
  case H5T_CONV_FREE:
    printf("Finalizing conversion function...\n");
    break;
  default:
    break;
  }
  return retval;
}
```

Listing 2.20 – User-defined datatype conversion – setup.

```c
int main() {
  hid_t src = H5Tcreate(H5T_COMPOUND, sizeof(struct src_t));
  H5Tinsert(src, "a", HOFFSET(struct src_t, a), H5T_NATIVE_UINT32);
  H5Tinsert(src, "b", HOFFSET(struct src_t, b), H5T_NATIVE_FLOAT);
  hid_t dst = H5Tcreate(H5T_COMPOUND, sizeof(struct dst_t));
  H5Tinsert(dst, "b", HOFFSET(struct dst_t, b), H5T_IEEE_F32LE);
  H5Tregister(H5T_PERS_SOFT, "src_t->dst_t", src, dst, &convert);
  struct src_t buf[] = {
    {1, 1.0} , {2, 2.0}, {3, 3.0}, {4, 4.0} , {5, 5.0} };
  H5Tconvert(src, dst, 5, buf, NULL, H5P_DEFAULT);

  H5Tunregister(H5T_PERS_SOFT, "src_t->dst_t", src, dst, &convert);
  H5Tclose(dst);
  H5Tclose(src);
  return 0;
}
```

Listing 2.21 – User-defined datatype conversion – invoked directly.

The second phase, `H5D__typeinfo_init_phase2()`, checks if selection I/O can be used and if the (expected) size of the conversion buffer and background buffer are sufficient to perform the conversion with the given selection.

The third and final phase, `H5D__typeinfo_init_phase3()`, allocates a type conversion buffer and a background buffer of appropriate size for each dataset that requires them and did not have them provided through transfer properties.

After the type info for the transfer is fully initialized, the write proceeds through a write callback specific to the layout of the target dataset(s). For example, for the program in Listing 2.22, the dataset's layout is `H5D_CONTIGUOUS`, and so the write callback is `H5D__contig_write()`. To gather elements from the user-provided buffer for contiguous writes to disk, `H5D__scatgath_write()` is invoked. Type conversion is then performed in the provided buffers right before the data is scattered to the file.

2.5.3. Datatype metadata & value encodings

In this section, we describe the datatype metadata retained in the HDF5 library and files, how the values of certain non-fixed-size datatypes are represented and encoded, and how the datatype affects where encoded values are stored in the HDF5 file.

Overview of Datatype Metadata Storage An HDF5 file is stored on disk as a global heap and data objects (among other things outside the scope of this section). Each data object has an 'object header' and 'object information.' The object information is the user's stored data, e.g., scientific measurements. The object header contains 'messages' which describe metadata regarding the information object.

Datatypes exist on disk as Datatype Messages according to the HDF file format. A Datatype Message contains information about the datatype's size, class, and class-specific properties.

```c
#include "common.h"
herr_t convert_float(hid_t src_id, hid_t dst_id, H5T_cdata_t *cdata,
    size_t nelmts, size_t buf_stride, size_t bkg_stride, void *buf,
    void *bkg, hid_t dxpl)
{
  herr_t retval = EXIT_SUCCESS;
  switch (cdata->command)
  {
  case H5T_CONV_INIT:
    printf("Initializing float conversion function...\n");
    break;
  case H5T_CONV_CONV:
    printf("Converting floats...\n");
    for (size_t i = 0; i < nelmts; ++i)
      ((float*) buf)[i] = ((double*) buf)[i];
    break;
  case H5T_CONV_FREE:
    printf("Finalizing float conversion function...\n");
    break;
  default:
    break;
  }
  return retval;
}

int main() {
  hid_t src = H5Tcopy(H5T_NATIVE_DOUBLE);
  hid_t dst = H5Tcopy(H5T_NATIVE_FLOAT);
  H5Tregister(H5T_PERS_SOFT, "double->float", src, dst, &convert_float);
  double buf[] = {1.0, 2.0, 3.0, 4.0, 5.0};

  hid_t file_id = H5Fcreate("float_conv.h5", H5F_ACC_TRUNC, H5P_DEFAULTx2);
  hid_t space_id = H5Screate_simple(1, (const hsize_t[]) {5}, NULL);
  hid_t dset_id = H5Dcreate(file_id, "dset", dst, space_id, H5P_DEFAULTx3);
  H5Dwrite(dset_id, src, space_id, H5S_ALL, H5P_DEFAULT, buf);
  H5Tunregister(H5T_PERS_SOFT, "double->float", src, dst, &convert_float);
  H5Fclose(file_id);
  H5Dclose(dset_id);
  H5Tclose(dst);
  H5Tclose(src);
  return 0;
}
```

Listing 2.22 – User-defined datatype conversion – invoked implicitly during write.

Transient datatypes are not written to disk unless they are used to create attributes or datasets. In this case, the object header for the attribute or dataset will contain a Datatype Message describing the datatype.

Recall that committed datatypes must be linked into the file hierarchy. A committed datatype is stored on disk as an object header with a Datatype Message. If a committed datatype is used in a dataset or an attribute, then the Datatype Message for that dataset or attribute contains a pointer to the original committed datatype's Datatype Message - the information is not duplicated.

Datatype Encoding Datatype encoding is the process of converting an in-memory datatype object (`H5T_t`) to a series of bytes which may be written to storage as a Datatype Message. This process may be invoked explicitly by `H5Tencode()/H5Tdecode()` or implicitly when writing objects with a datatype to storage.

Figure 2.29 shows the library's datatype encoding process. Some internal routines require a valid `H5F_t` file struct but do not modify or depend on a real external file. To support datatype encoding/decoding without a real associated file, `H5F_fake_alloc()` creates a 'fake' file struct with no counterpart in storage. The size of the message to create is determined by `H5O_msg_raw_size()`, which invokes the Datatype Message class's 'size' callback, `H5O__dtype_size()`. Similarly, the generic `H5O_msg_encode()` invokes the Datatype Message class's encode callback, `H5O__dtype_encode()`. This encode callback populates the provided buffer with information from the datatype. Suppose the datatype is non-atomic; recursive calls to `H5O__dtype_encode_helper()` populate each component datatype. For compound types, a separate recursive call encodes each member field. A single recursive call is used to encode the base type for arrays, variable-length types, and enums.

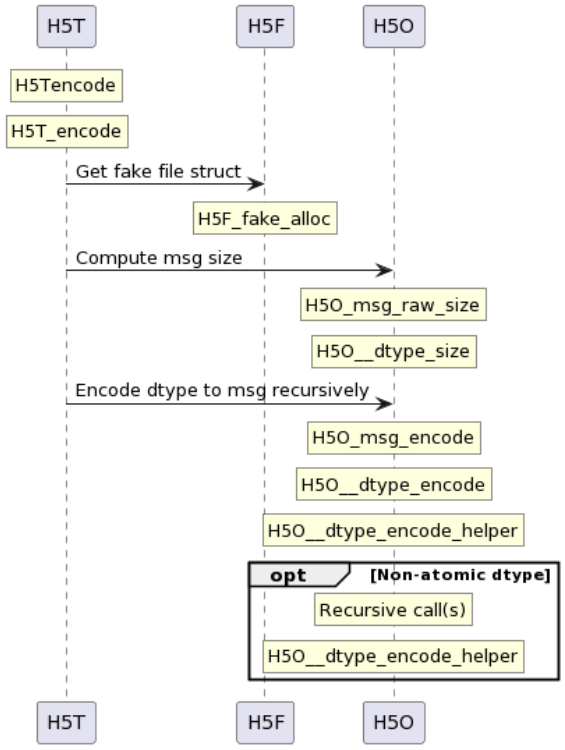


Figure 2.29. – Datatype encoding process

Datatype storage Data with a simple atomic datatype is stored in the object information section of the data object. Because elements of more complex datatypes may have an indeterminate or variable size, they must be stored differently. Variable-length and region reference data are stored in the file's global heap. Object reference data is stored as the offsets necessary to read from the object header of the referenced object. Compound datatype elements are stored as a contiguous stream of the structure's items, each formatted according to its datatype.

2.5.4. Summary

In this tour, we covered the different categories of datatypes, how they are represented and handled by the library, and how objects use them. We reviewed how datatypes are encoded for storage on disk and how data of different datatypes is stored in the file.

The ability for the user to define their own datatypes, composite datatypes, and datatype conversion functions is another example of the extensibility and customizability of the HDF5 library, allowing users to optimize nearly every step in the I/O process.

2.6. Getting down and dirty with caches

Throughout this tour, we describe the different user-controllable caching and buffering facilities. Figure 2.30 shows an overview of these facilities.

There are four architectural elements in the native VOL that implement caching or buffering functionality.

- **Metadata cache:** We have encountered the metadata cache in Sections 2.2 and 2.3.
- **Contiguous data cache:** Frequently accessed contiguous dataset regions can be held in memory to minimize small I/O for datasets with contiguous storage layouts.
- **Chunk cache:** Frequently accessed chunks can be held in memory to minimize chunk I/O for datasets with chunked storage layouts.
- **Page buffer:** Since HDF5 library version 1.10.0, file space can be managed and I/O operations performed in fixed-size pages. This so-called 'paged aggregation' or 'paged allocation' scheme is most effective when combined with a page buffer.

The cache vs. buffer terminology in the native VOL is a little murky. The term 'cache' is usually applied to a data structure that, in its implementation, uses several slots to which specific entries are assigned and which is controlled by a hash function that might have a degree of associativity. Typically, there is also an entry eviction or prefetch policy defined. The metadata and chunk caches are caches in that sense. The contiguous data cache (or sieve buffer), not so much. The native VOL caches and the page buffer gather internal usage statistics, which can be used to assess their effectiveness for particular applications.

During this tour, we take a *qualitative* look at several examples where we will see the different facilities in action. This is, however, not the place to quantify the actual performance impact. This is highly problem size and access pattern-dependent, and the significant influence of the operating system's memory page cache cannot be overestimated.

2.6.1. “Raw" Data (RD) and Metadata (MD)

We already encountered the metadata cache in tours 2.2 and 2.3. Before looking at a specific example, let’s remind ourselves what, in this context, we mean by metadata versus raw data.

Metadata	Raw Data
■ Entries enumerated in `H5AC_type_t` ■ File pages filled with metadata	■ Contiguous regions of datasets ■ Dataset chunks ■ File pages filled with raw data

`H5AC_type_t` is defined in `H5ACprivate.h`, and, with a few exceptions, contains many of the metadata items found in the HDF5 file format specification [10] such as file superblock, symbol table nodes, local heaps, object headers, B-tree nodes, etc. If a paged allocation is used and a page buffer is present (see section 2.6.4), file space is managed in fixed-size pages containing either metadata or raw data. The page buffer is a memory region set aside to store a list of frequently accessed file pages.

2.6.2. The mighty metadata cache (MDC)

To simulate the impact of an improperly configured (or altogether absent) MDC, we perform a set of metadata-intensive operations and arbitrarily “throttle" the MDC via `H5Pset_mdc_config()` as shown in listing 2.23. The code creates and then deletes 50,000 subgroups in separate loops and then creates another 25,000 subgroups. Most of the metadata will be local heaps and symbol tables from all these groups. Running the code with the `throttle` argument curtails the metadata cache size to 1KiB. With the throttled configuration, the code runs more than five times slower than the default configuration. Reviewing the Callgrind profile, this increase is almost proportional to the increase in `H5C__flush_single_entry()` calls from `H5C__make_space_in_cache()` (429,136 to 1,873,837 with `throttle`).

2.6.3. Caching “raw" data

Contiguous data cache (a.k.a. data sieve buffer)

Data sieving [22] is a common technique to make a few large, contiguous requests to the file system, even if the user’s request consists of several small, non-contiguous accesses, e.g., point or hyperslab selections. The native VOL uses this technique for datasets with contiguous layouts. The definition of the so-called ‘raw data contiguous data cache’ can be found in `H5Dpkg.h` and is repeated in listing 2.24. `sieve_loc` is the offset of the sieve in the HDF5 file, `sieve_size` is the used size of the total `sieve_buf_size`, and `sieve_dirty` tracks the sieve buffer’s modification status. The sieve buffer size can be controlled at the file level(!) via `H5Pset_sieve_buf_size()`.

```c
#include "common.h"
void set_mdc_config(H5AC_cache_config_t *config)
{
    config->version                 = H5AC__CURR_CACHE_CONFIG_VERSION;
    config->rpt_fcn_enabled         = 0;
    config->open_trace_file         = 0;
    config->close_trace_file        = 0;
    config->evictions_enabled       = 1;
    config->dirty_bytes_threshold   = 1024;
    config->max_size                = 1024;
    config->min_size                = 1024;
    config->set_initial_size        = 0;
    config->epoch_length            = 50000;
    config->incr_mode               = H5C_incr__off;
}
int main(int argc, char** argv) {
    H5AC_cache_config_t mdc_config;
    hid_t fapl = H5Pcreate(H5P_FILE_ACCESS);
    if (argc > 1 && strcmp(argv[1], "throttle") == 0) {
        set_mdc_config(&mdc_config);
        H5Pset_mdc_config(fapl, &mdc_config);
    }
    hid_t file=H5Fcreate("group.1.h5",H5F_ACC_TRUNC,H5P_DEFAULT,fapl);
    hid_t group1 = H5Gcreate(file, "group1", H5P_DEFAULTx3);
    char data[16] = "subgroup XXXXX";
    for (size_t i = 0; i < 50000; ++i) {
        snprintf(data + 8, 6, "%05zu", i);
        H5Gclose(H5Gcreate(group1, data, H5P_DEFAULTx3));}
    for (size_t i = 0; i < 50000; i += 2) {
        snprintf(data + 8, 6, "%05zu", i);
        H5Gunlink(group1, data);}
    H5Gclose(group1);
    hid_t group2 = H5Gcreate(file, "group2", H5P_DEFAULTx3);
    for (size_t i = 0; i < 25000; ++i) {
        snprintf(data + 8, 6, "%05zu", i);
        H5Gclose(H5Gcreate(group2, data, H5P_DEFAULTx3));}
    H5Gclose(group2);
    H5Fclose(file);
    return 0;
}
```

Listing 2.23 – Tiny MDC.

```
typedef struct H5D_rdcdc_t {
    unsigned char *sieve_buf;
    haddr_t        sieve_loc;
    size_t         sieve_size;
    size_t         sieve_buf_size;
    bool           sieve_dirty;
} H5D_rdcdc_t;
```

Listing 2.24 – Raw data contiguous data cache (`rdcdc`!).

Chunk cache

The chunk cache is defined as struct `H5D_rdcc_t` in `H5Dpkg.h`, and the definition is too long to be reproduced here. Like most caches, it has a set of slots for cache entries, an auxiliary doubly-linked list to implement an eviction strategy based on LRU or similar, and performance counters for statistics.

To see the chunk cache in action and the side effects of an undersized chunk cache, we have created the example in listing 2.25. In this example, we update single elements on different chunks, forcing chunk cache evictions if the chunk cache cannot hold all chunks. The dataset has 1,048,576 32-bit integer elements allocated across four chunks. The default chunk cache size is 1MiB, which can hold a single chunk. In every loop iteration, we update an element on a different chunk, evicting the current chunk. Passing the `cache-all` argument to the program increases the chunk cache to 4MiB, which can hold all four chunks in the chunk cache without evictions. The Callgrind profiles' main difference is 100 calls to `H5D__chunk_cache_prune()` with the default (undersized) chunk cache, which we expected. Because of the OS's page cache, the actual performance difference is negligible for the small chunk sizes in this example. However, this would not be true for larger datasets, chunks, other access patterns, and system loads.

2.6.4. Page buffering

In HDF5 library release 1.10.0, a new file space management strategy, *paged aggregation*, that aggregates small metadata and raw data allocations into constant-sized, well-aligned pages was introduced [2]. Paged aggregation combined with *page buffering* [1] allows more efficient I/O accesses.

To see the paged aggregation/page buffering combination in action, we have modified listing 2.25 and updated the file creation and access property lists to arrive at listing 2.26. We have updated the behavior of the `cache-all` argument to bump the page buffer size from 1MiB to 8MiB. The latter can accommodate all file pages. Again, looking at the differences in the Callgrind profiles, the differences are clear to see. A good measure of the effectiveness of paged aggregation and page buffering is to look at the number of `H5PB[read,write]()` and `pwrite()` calls.

`H5PBwrite()` writes data into the page buffer. If the page exists in the cache, update it; otherwise, read it from the disk, update it, and insert it into the cache.

`H5PBread()` reads in the data from the page containing it if it exists in the page buffer; otherwise, it reads the page through the VFD.

The number of `pwrite()` calls indicates the I/O traffic the OS sees from the application.

```c
#include "common.h"
int main(int argc, char** argv)
{
    hid_t file = H5Fcreate("data.5.h5", H5F_ACC_TRUNC, H5P_DEFAULTx2);
    hid_t fspace = H5Screate_simple(1, (hsize_t[]){_1MIB}, NULL);
    hid_t dcpl = H5Pcreate(H5P_DATASET_CREATE);
    H5Pset_chunk(dcpl, 1, (hsize_t[]){_256KIB});
    hid_t dapl = H5Pcreate(H5P_DATASET_ACCESS);
    if (argc > 1 && strcmp(argv[1], "cache-all") == 0)
        H5Pset_chunk_cache(dapl, H5D_CHUNK_CACHE_NSLOTS_DEFAULT, _4MIB,
            H5D_CHUNK_CACHE_W0_DEFAULT);
    hid_t dset = H5Dcreate(file, "data", H5T_NATIVE_INT, fspace,
        H5P_DEFAULT, dcpl, dapl);
    hid_t mspace = H5Screate_simple(1, (hsize_t[]){1}, NULL);
    H5Sselect_elements(mspace, H5S_SELECT_SET, 1, (hsize_t[]){0});

    for (int i = 0; i < 100; ++i)
    {
        H5Sselect_elements(fspace, H5S_SELECT_SET, 1,
            (hsize_t[]){(i%4)*_256KIB+i});
        H5Dwrite(dset, H5T_NATIVE_INT, mspace, fspace, H5P_DEFAULT, &i);
    }
    H5Sclose(mspace);
    H5Pclose(dapl);
    H5Pclose(dcpl);
    H5Dclose(dset);
    H5Sclose(fspace);
    H5Fclose(file);
    return 0;
}
```

Listing 2.25 – Undersized chunk cache.

```c
#include "common.h"
int main(int argc, char** argv)
{
    hid_t fcpl = H5Pcreate(H5P_FILE_CREATE);
    H5Pset_file_space_strategy(fcpl, H5F_FSPACE_STRATEGY_PAGE, 1, _4KIB);
    H5Pset_file_space_page_size(fcpl, _4KIB);
    hid_t fapl = H5Pcreate(H5P_FILE_ACCESS);
    if (argc > 1 && strcmp(argv[1], "cache-all") == 0)
        H5Pset_page_buffer_size(fapl, _8MIB, 0, 0);
    else
        H5Pset_page_buffer_size(fapl, _1MIB, 0, 0);

    hid_t file = H5Fcreate("data.6.h5", H5F_ACC_TRUNC, fcpl, fapl);
    hid_t fspace = H5Screate_simple(1, (hsize_t[]){_1MIB}, NULL);
    hid_t dcpl = H5Pcreate(H5P_DATASET_CREATE);
    H5Pset_chunk(dcpl, 1, (hsize_t[]){_256KIB});
    hid_t dapl = H5Pcreate(H5P_DATASET_ACCESS);
    if (argc > 1 && strcmp(argv[1], "cache-all") == 0)
        H5Pset_chunk_cache(dapl, H5D_CHUNK_CACHE_NSLOTS_DEFAULT, _4MIB,
            H5D_CHUNK_CACHE_W0_DEFAULT);

    hid_t dset = H5Dcreate(file, "data", H5T_NATIVE_INT, fspace,
        H5P_DEFAULT, dcpl, dapl);
    hid_t mspace = H5Screate_simple(1, (hsize_t[]){1}, NULL);
    H5Sselect_elements(mspace, H5S_SELECT_SET, 1, (hsize_t[]){0});

    for (int i = 0; i < 100; ++i)
    {
        H5Sselect_elements(fspace, H5S_SELECT_SET, 1,
            (hsize_t[]){(i%4)*_256KIB+i});
        H5Dwrite(dset, H5T_NATIVE_INT, mspace, fspace, H5P_DEFAULT, &i);
    }
    H5Sclose(mspace);
    H5Pclose(dapl);
    H5Pclose(dcpl);
    H5Dclose(dset);
    H5Sclose(fspace);
    H5Fclose(file);
    return 0;
}
```

Listing 2.26 – Paged allocation and page buffering.

The call counts without and with the `cache-all` argument are shown in parentheses.

- `H5PBwrite()` (114 / 18)
- `H5PBread()` (96 / 0)
- `pwrite()` (102 / 6)

The drop in call counts is significant. Again, depending on the problem size, the effectiveness of the OS page cache will intervene, and the actual performance difference between the two scenarios may not be near what the counts suggest.

2.6.5. Summary

During this tour, we looked at four user-controllable caching and buffering facilities for metadata and raw data in the native VOL. They are effective on paper, and the default configurations might often be sufficient. Ideally, these features would be adaptive and self-tuning since their inclusion in any code base is fraught with portability and maintenance issues. Finally, three of the four facilities also capture statistics that can be used to assess their effectiveness in real-life scenarios.

2.7. Multithreading

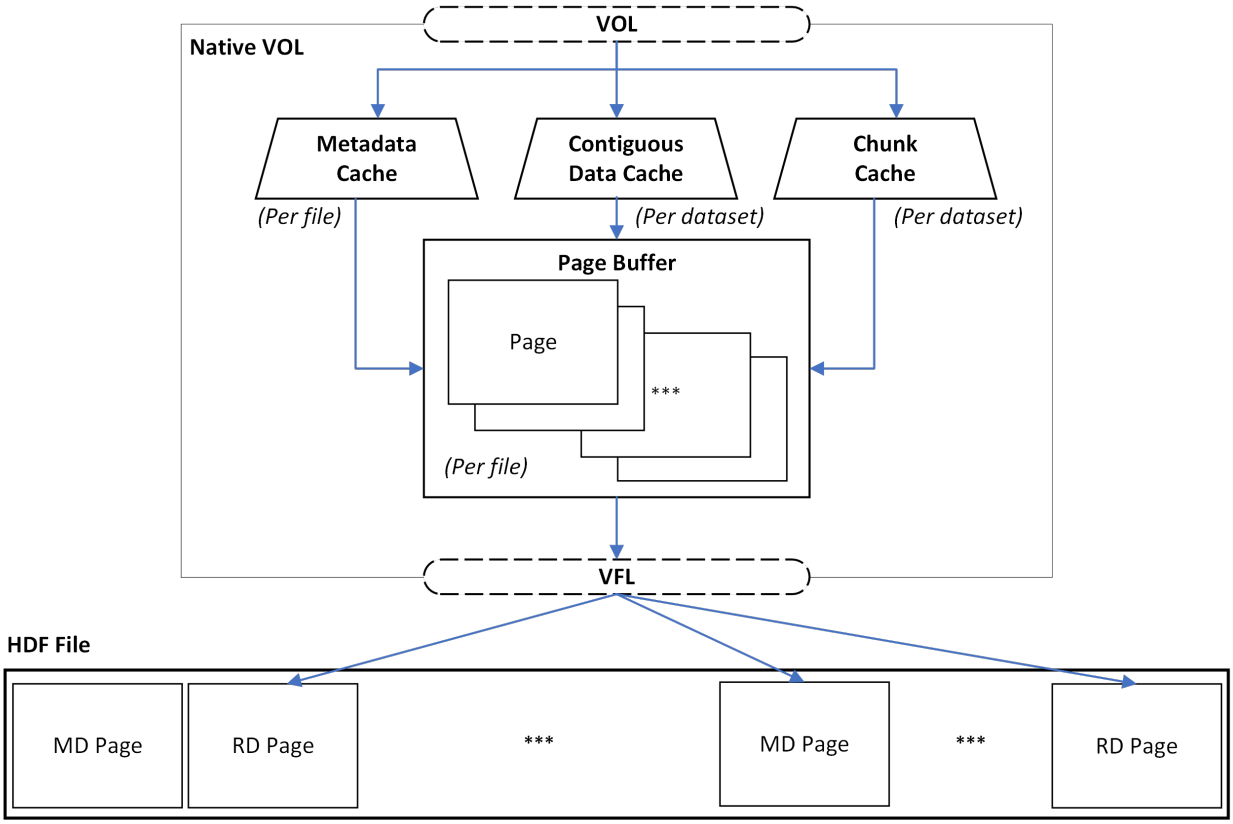


Figure 2.30. – Native VOL caching and buffering.

In Listing 2.27, a simple application is shown that uses three threads to create three datasets (one per thread) in an HDF5 file. The order in which the datasets are created is “randomized" by calls to `sleep()` function with a random input inside the `create_dataset()` function. Except for the presence of the `H5TS` package and Pthreads library calls, the Callgrind profile is indistinguishable from what we have seen in other tours. This seamlessness is achieved by “sleight of hand" and expectation management: The library provides a thread-safe implementation but not a full-blown multi-thread concurrent implementation. Essentially, four elements can achieve this:

- A modified global library initialization variable
- A global thread initialization variable
- A global key for per-thread error stacks
- A global structure and key for thread cancellation prevention.

For a sample run, in the Callgrind profile, only seven `H5TS` APIs appear (call counts in parentheses). The first three are infrastructure:

- `H5TS_pthread_first_thread_init()` (1)
- `H5TS_tid_init()` (1)
- `H5TS_key_destructor()` (9).

The remaining four represent the steady state:

- `H5TS_mutex_[lock,unlock]()` (90 each)
- `H5TS_cancel_count_[inc,dec]()` (89 each).

To explain this behavior, in section 2.7.1, we review the thread-safety related changes to `H5private.h` and core functions of the `H5TS` package.

2.7.1. The implementation of thread-safety in the HDF5 library

On UNIX-like systems, the implementation uses POSIX threads; Windows threads are used under Windows. Note that we use Pthreads in the examples for illustration.

A good, albeit slightly outdated, overview of the thread-safety implementation can be found in [15].

The HDF5 library must be built with the `-enable-threadsafe` option for multithreaded applications to behave correctly. This defines the `H5_HAVE_THREADSAFE` macro, modifying library global infrastructure in `H5private.h`. The most notable changes are the following:

1. The global variables `H5_lib[init,term]_g` were replaced by a struct and protected by a mutex, see listing 2.28.
2. The global variable `H5TS_first_init_g` of type `pthread_once_t` is used to allow only the first thread in the application process to call an initialization function using `pthread_once()`. Any thread’s subsequent calls to `pthread_once()` are disregarded. The call sets up the mutex in the global structure `H5_g` via an initialization function `H5TS_pthread_first_thread_init()`.
3. A global thread-managed key `H5TS_errstk_key_g` is used to allow pthreads to maintain a separate error stack (of type `H5E_t`) for each thread.

```c
#include "common.h"

struct targs {
    unsigned count;
    hid_t file;
};

void* create_dataset(void* args)
{
    struct targs* ptr = (struct targs*)args;
    printf("Hello from the new thread! %d\n", ptr->count);
    sleep(rand() % 2);
    hid_t fspace = H5Screate_simple(1, (hsize_t[]){10*(ptr->count+1)},
        NULL);
    char name[10];
    snprintf(name, 10, "dataset%d", ptr->count);
    hid_t dataset = H5Dcreate(ptr->file, name, H5T_NATIVE_INT,
        fspace, H5P_DEFAULTx3);
    H5Dclose(dataset);
    H5Sclose(fspace);
    return NULL;
}

int main() {
    hid_t file = H5Fcreate("mt.h5", H5F_ACC_TRUNC, H5P_DEFAULTx2);

    pthread_t t1, t2, t3;

    struct targs batch[3] = {{1, file}, {2, file}, {3, file}};

    pthread_create(&t1, NULL, (void*)create_dataset, (void*)&batch[0]);
    pthread_create(&t2, NULL, (void*)create_dataset, (void*)&batch[1]);
    pthread_create(&t3, NULL, (void*)create_dataset, (void*)&batch[2]);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);

    H5Fclose(file);
    return 0;
}
```

Listing 2.27 – A multithreaded application using POSIX threads (Pthreads).

```
typedef struct H5_api_struct {
    H5TS_mutex_t init_lock;
    bool         H5_libinit_g;
    bool         H5_libterm_g;
} H5_api_t;
```

Listing 2.28 – Modified global library initialization variable.

```

#define FUNC_ENTER_API_THREADSAFE \
    H5_FIRST_THREAD_INIT          \
    H5_API_UNSET_CANCEL           \
    H5_API_LOCK

#define FUNC_LEAVE_API_THREADSAFE  \
    H5_API_UNLOCK                  \
    H5_API_SET_CANCEL

```

Listing 2.29 – Modified `FUNC_[ENTER,LEAVE]` macros in `H5private.h`.

4. To prevent thread cancellations from killing a thread while it is in the library, we maintain per-thread information about the cancellability status of the thread before it enters the library so that we can restore that same status when the thread leaves the library. The status is preserved using a structure (of type `H5TS_cancel_t`) which maintains the cancellability state of the thread before it entered the library and a count (which works very much like the recursive lock counter) that keeps track of the number of API calls the thread makes within the library. For details, see `H5TS.c`.
5. The `FUNC_ENTER` and `FUNC_LEAVE` macros were modified as shown in listing 2.29 to (un-)lock the (re-)store the thread cancellation status.

2.7.2. Summary

At the time of this writing, the HDF5 library provides “only" a thread-safe implementation but not a full-blown multi-thread concurrent implementation. Given this narrower scope, this is achieved at a low cost and low implementation complexity.

3. Architectural Overview

Software architecture is the foundation of a software system that guides its development, ensuring it is scalable and reliable. It helps identify risks early on, simplifies updates and maintenance, aligns technical aspects with business goals, and promotes effective team coordination and communication. A well-defined architecture makes the system less complex, easier to maintain, less costly and increases productivity and flexibility. This chapter aims to provide a deeper understanding of these topics in relation to the HDF5 library. It also seeks to document the existing architectural issues and their impact on the HDF5 library while exploring ideas to address issues and enhance its functionality without breaking it.

3.1. Why software architecture?

Software architecture plays a crucial role in the development and maintenance of software systems for several reasons [17]:

- **Foundation for System Design:** Software architecture provides the fundamental structure of a software system. It outlines the software's components, their relationships, and how they interact, serving as a blueprint for both the system and the project developing it.
- **Facilitates Scalability and Performance:** Good architecture makes it easier to scale the software up or out to meet increasing demand or improve performance. It ensures the system can handle the growth of user, data, or transaction volume without significant rework.
- **Improves Quality and Reduces Risks:** Well-defined architecture helps identify potential risks and technical challenges early in the development process, allowing for proactive mitigation. It also sets standards for quality in coding, which can lead to a more stable and reliable product.
- **Enhances Maintainability and Flexibility:** A clear and modular architecture simplifies maintenance and updates. It allows different system parts to be updated or replaced without affecting the rest of the system, thereby supporting flexibility and adaptability to changing requirements or technologies.
- **Facilitates Team Coordination and Communication:** In large projects, having a clear architecture helps organize the work among various teams. It provides a common language and understanding, helping teams to coordinate and communicate more effectively.
- **Aligns with Business Goals:** Software architecture can align the technical aspects of the project with business goals. It helps in making strategic decisions about which technologies to use, how to allocate resources, and how to prioritize different aspects of the project.
- **Optimizes Cost and Resource Usage:** The development team can use resources more efficiently by planning the architecture. It can prevent over-engineering in some areas while focusing efforts where they are most needed, thus optimizing developmental and operational costs.

What happens without software architecture? [17]

In the absence of a well-defined software architecture, various challenges and issues can arise that impact the development process, the quality of the software, and its long-term viability:

- **Increased Complexity:** Without a clear architectural plan, the software can become a patchwork of ad-hoc solutions. This often increases complexity, making the system harder to understand, modify, and maintain.
- **Scalability and Performance Issues**: Without a proper architecture, the system may not scale well with increased load or user numbers. Performance bottlenecks are often not apparent until the system is under stress; at that point, they can be costly and time-consuming.
- **Poor Quality and Reliability:** The absence of a structured architecture often leads to inconsistencies in design and coding practices. This can result in a less reliable system with more bugs and issues, impacting the user experience and the software's credibility.
- **Difficult Maintenance and Upgrades:** Changes and maintenance become challenging in systems without clear architecture, as the impact of modifications is hard to predict. This can lead to a "domino effect" where changes in one part of the system unexpectedly affect other parts.
- **Reduced Productivity:** Developers working on a poorly structured system often find it challenging to locate and understand parts of the codebase. This can significantly slow development and increase the likelihood of introducing new errors when making changes.
- **Higher Costs:** In the long run, the absence of an exemplary architecture can lead to higher costs. More resources are needed to manage the complexities, fix issues, and implement changes. The cost of rewriting or significantly refactoring the software can also be substantial.
- **Difficulty in Team Coordination:** Without a clear architectural roadmap, coordinating a team of developers becomes challenging. It can lead to duplicated efforts, inconsistent implementations, and difficulty integrating different system parts.
- **Limited Flexibility and Adaptability:** A software without a planned architecture may become rigid and inflexible, making it hard to adapt to new requirements, technologies, or market changes. This can hinder the software's ability to stay competitive and relevant.

Where does the HDF5 library architecture stand on this?

3.2. Architectural Problems

The purpose of this section is to collect and document known *architectural* issues in the HDF5 library. For now, this is more of a placeholder, but not because there aren't any known issues. It's hard to figure out where to start. So, watch this space!

Perhaps a good starting point would be to approach this topic from the perspective of flexibility [12] or extensibility [24]. How has the HDF5 library architecture changed over 25 years? What drove those changes? Were they graceful or at least successful?

> "Our goal in this book is to investigate how to construct computational systems so that they can be easily adapted to changing requirements. One should not have to modify a working program. One should be able to add to it to implement new functionality or to adjust old functions for new requirements. We call this *additive programming*. We explore techniques to add functionality

> to an existing program without breaking it. Our techniques do not guarantee that the additions are correct: the additions must themselves be debugged; but they should not damage existing functionality accidentally." [12]

What are the “multiplicative" elements in the HDF5 library architecture?

Looking at the partial chronology in Appendix B, how many of those features have made the HDF5 library a better product?

What is the scale of accumulated technical debt in the code base, and what fraction is due to poor architecture?

4. Reference

This chapter provides an in-depth explanation of the HDF5 library implementation and design, particularly emphasizing how it relates to the HDF5 abstract data model components that were briefly touched upon in previous chapters. The chapter's primary objective is to provide a detailed and informative description of the systemic, structural, and orderly aspects of each HDF5 module and its interconnection. The chapter aims to provide a complete understanding of the library's architecture and design decisions, enabling a better understanding of the software architecture and design decisions that have been made.

4.1. Modules

An overview of HDF5 library modules' locations is shown in Figure 4.1. The organizing principle is the VOL architecture, a set of interfaces for defining mappings between HDF5 virtual objects and storage primitives. When attempting to map the HDF5 library API to the VOL interface, inevitably, HDF5 library API idioms that do not deal with mappings of HDF5 virtual objects to storage are left "homeless." This puts the subset of HDF5 library API calls in focus that cannot be mapped to the VOL interface. We refer to this part of the HDF5 library API (and the modules in which they are implemented) as non-VOL API calls. In the diagram, the top portions of the blue boxes "split" the HDF5 library API. While VOL is an interface, the non-VOL collection was never intended as an interface, giving it a less polished appearance.

The subset of modules of the pre-VOL HDF5 library that deal with mappings of HDF5 virtual objects to storage make up the core of the native VOL connector. Figure 2.16 offers a blown-up view of the native VOL connector. Pre-VOL HDF5 library extension interfaces, such as the virtual file layer (VFL) shown in yellow, are retained but are specific to the native VOL.

Both the modules that make up the native VOL connector and those that support non-VOL API calls share a set of HDF5 library infrastructure modules (shown in the gray box).

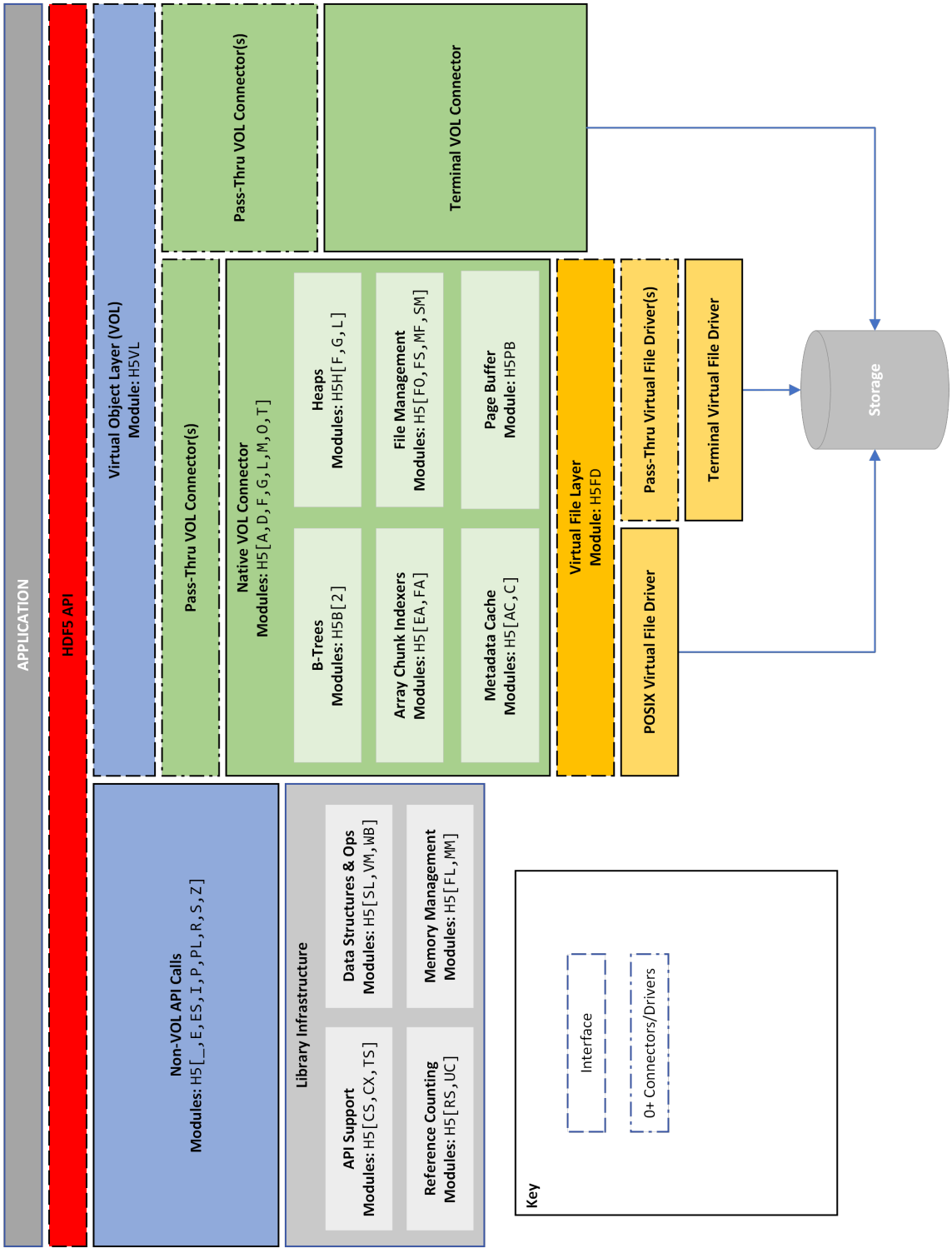


Figure 4.1. – The "location" of HDF5 library modules.

4.1.1. File Memory Allocation (H5MF)

The file memory management module H5MF is one of the major clients of the Free Space Manager (FSM) module H5FS. Many file memory management routines act as wrappers around corresponding FSM operations. H5MF also manages some higher-level memory processes, such as paged aggregation (if enabled) and free space aggregators (if paged aggregation is disabled).

- Allocation Types

  H5MF uses the type of allocation requested to determine which type of FSM to create, open, or use for each file memory request. Allocation type indicates what type of low-level file element the allocated memory block is for. The allocation types are superblock, B-tree, raw data (DRAW), global heap, local heap, and object header. All of these allocation types besides 'raw data' are metadata.

- Free Space Aggregators

  If the filespace management strategy is H5F_FSPACE_STRATEGY_FSM_AGGR or H5F_FSPACE_STRATEGY_AGGR, then small memory allocation requests are handled by two aggregators - one for raw data, and one for metadata. Aggregators are represented in memory with the H5F_blk_aggr_t struct. Each aggregator maintains an 'allocation block' used to satisfy small memory requests of the appropriate type. This grouping of small sections of data into larger blocks prevents tiny, scattered accesses, improving performance substantially when compared to unmanaged filespace allocation. If an aggregator does not have enough space in its current allocation block to satisfy an allocation request, the file is extended to satisfy the request. This extension also occurs whenever an allocation request exceeds the size of an allocation block.

  While the allocation blocks created by the aggregators are an improvement over naive allocation, the allocation blocks have some issues themselves. The blocks are not page aligned, and the allocation blocks themselves end up scattered randomly through the file. These issues are addressed by paged aggregation.

- Paged Aggregation

  If the filespace management strategy is H5F_FSPACE_STRATEGY_PAGE, paged aggregation is enabled, and the raw data/metadata aggregators are disabled. They are replaced by a set of 'small' and 'large' FSMs. These FSMs group allocations into pages, making the aggregators unnecessary. The small FSMs track free sections which are smaller than the filespace page size, and attempt to satisfy small allocation requests. If a small FSM cannot satisfy an allocation, it requests additional memory from a large FSM. A large FSM tracks free sections that are equal to or larger than the filespace page size, and attempts to satisfy large allocation requests. If a large FSM cannot satisfy an allocation, it requests additional memory from the VFD.

  For a file with a contiguous address space, the default behavior is to use two small FSMs (one for raw data and one for metadata) and one large FSM (for all large allocations, regardless of whether they are raw data or metadata). For a file with non-contiguous address space, it is possible to have up to 6 small FSMs and 6 large FSMs. These correspond to the six file space types: raw data and five types of metadata.

- Persistent Free Space Tracking and Self-Referential FSMs

  Filespace allocations and their associated FSMs can be divided into three categories: allocations and FSMs for raw data, allocations and FSMs for metadata other than FSMs, and allocations and FSMs

which handle memory for FSMs themselves. This last category is called "self-referential FSMs". Non-self-referential FSMs are stored in the metadata cache's raw data FSM region. It may seem confusing that some metadata FSMs are stored in a cache region named for raw data FSMs, but this is because they may be treated like raw data FSMs by the cache, while self-referential FSMs require special handling.

If persistent free space tracking is enabled, FSMs are written to the file. Before they are written to file, they must be 'settled' to minimize filespace used, remove redundant or outdated information from the file, and allow for self-referential FSMs to be written consistently and safely. Settling non-self-referential FSMs involves trimming the End of Address (EOA), deallocating filespace for old non-self-referential FSMs and the old FSM superblock extension message, and allocating new filespace for the current batch of non-self-referential FSMs and a new FSM superblock extension message. Non-empty (non-self-referential) FSMs are not written to the file since they contain no useful information, so no space is allocated for them. This settling process only involves deallocating and allocating memory, with all the actual data writes performed through the metadata cache later.

Self-referential FSMs must be settled after all non-self-referential FSMs. Self-referential-FSMs are resolved in a slightly modified way to avoid two potential infinite loops. The first infinite loop would be allocating space for an FSM that is eventually found to contain no free space sections. This space would then need to be de-allocated again (because empty FSMs are not written to the file), potentially triggering an infinite loop. The second possible infinite loop would be allocating space for a free section info block and, in the process, increasing the size of that free section info block. This would necessitate reallocating that info block, potentially leading to an infinite loop. These loops are avoided by changing the settling process for self-referential FSMs in two ways. First, empty self-referential FSMs are written to the file, unlike empty non-self-referential FSMs. Second, free section info blocks for self-referential FSMs can be oversized.

Settling self-referential FSMs requires saving the EOA to file. Because the split and multi-file drivers work with multiple files, resulting in numerous valid EOAs, persistent filespace management is not supported with those VFDs.

4.1.2. Free Lists (`H5FL`)

The library's internal free lists module `H5FL` is designed to allow the library to make as few calls to system memory functions as possible in order to improve performance. The primary interface exposed by this module to the rest of the library is a set of macros to define, add to, and remove free sections from a free list. Other modules in the library use these macros to create and operate on free lists associated with a particular struct in that client module. For example, the page buffer module uses a separate free list to handle memory for page buffers (`H5PB_t`) and page buffer entries (`H5PB_entry_t`). These free lists, as an internal library optimization mechanism, are not tied to a particular file but to the library as a whole. The free list module sits below other modules that manage memory allocations at the file level, such as `H5FS` and `H5MF`.

Implementing the free list interface through macros allows the free list operations to be silently replaced with system-free and allocate routines if free lists are disabled at build time. The option to disable free lists was implemented to simplify memory allocation debugging.

- In-memory objects

  The library has four main types of free lists: 'regular' free lists for single internal data structures, array free lists for arrays of internal data structures, block free lists for arbitrarily sized blocks of bytes, and

factory free lists for fixed-size blocks of bytes. Each of these free list types has its own set of structs used to represent it in the library.

Each free list type has a head struct (`H5FL_<type>_head_t`) that acts as the head of the free list. This head points to a list of free blocks that are all of the same fixed size. Each free list type has a node struct (`H5FL_<type>_node_t`) that acts as the elements of the linked free list.

A fifth free list type, sequence free lists, is a variation on array free lists. Sequence free lists store free blocks for unbounded arrays of internal data structures. Sequence free lists are represented as a block free list with an additional field for sequence element size.

- Block and Array Free Lists

  Block and array free lists, due to supporting arbitrary size blocks of bytes, are slightly more complicated than other free list types. They are implemented using three architectural elements. The first element is lists of free blocks of a given size. Each element of this list is represented as an instance of `H5FL_<type>_list_t`. Each element in the list stores information about each free block/array. The second element is nodes, which sit either in a priority queue (for blocks) or in an array (for arrays). Each node is associated with a free list, and each node stores pointers to adjacent nodes. Nodes in the priority queue/array are represented as instances of `H5FL_<type>_node_t`. The third element is a pointer to the head of the priority queue/array, represented by `H5FL_<type>_head_t`. Figure 4.2 shows the structure of a block-free list.

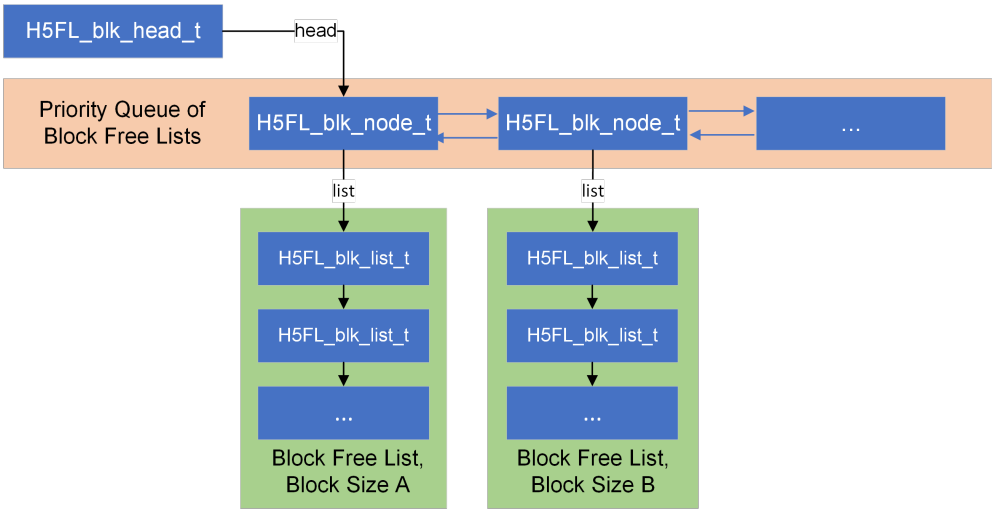


Figure 4.2. – A block free list

When a memory allocation for a block or array is requested from a free list, the collection of free lists for various block/array sizes is searched for the appropriate free list. If a free list of the proper size is found, a block of memory from that free list is used. In the case of a free block list, the free list that was found is moved from its original position to the head of the queue, speeding up subsequent accesses.

- Garbage Collection

  A major benefit of library-handled free lists is that they enable library-handled garbage collection. Garbage collection is the process by which the library returns library-managed free space to the filesystem.

Garbage collection functionality is exposed to applications through `H5garbage_collect()`. This is one of many ways the library exposes the memory and performance tradeoff to applications - less frequent garbage collection will use more memory but involve fewer system calls. In contrast, frequent garbage collection keeps memory overhead low but may involve slow allocations.

When a request for memory allocation (through `H5MM`) cannot be satisfied by the system, the library runs garbage collection and retries the system memory allocation call before throwing an error. This avoids a scenario where the library has memory available to it through free lists, but requests to the system for memory allocation fail due to the system being unaware of this memory.

- Performance Considerations

  Free lists are not created for all library types that require memory allocation. Because the free list infrastructure imposes some overhead in terms of memory and performance, only types that are expected to be allocated and deallocated many times during typical library operation are managed through a free list. Other types have their memory allocated and freed more directly through the `H5MM` module.

4.1.3. API Contexts (`H5CX`)

- API Contexts vs. Global Variables

  The primary goal of API contexts is to provide a way to get and set values from many different library layers during an API routine without the performance penalty and complexity of extra parameters and without the tight inter-module coupling and un-traceability of global variables.

  Each API routine creates an API context containing fields to accomplish this goal. These fields control aspects of the operation, deciding things such as which VOL to use, or whether a metadata read is collective. Each of the fields in a context is associated with its own accessor ("get/set") routines. At the end of the API routine, the context is released. The temporary lifetime and accessor-based usage of API contexts avoid the shortcomings of global variables while avoiding the performance penalty of passing around many extra parameters.

- In-memory objects

  An API context is represented by the `H5CX_t` struct. This struct has at least one member for each field accessible through the API context interface. Four distinct kinds of context fields are described in Table 4.1. Cached fields and return-only fields are paired with an additional field `<field_name>_valid`, used by macros that interact with that property.

- Reentrancy and The Context Stack

  Some API routines, such as `H5Literate()`, involve user-defined callbacks, which may themselves call API routines. Thus, API contexts must be re-entrant safe. This is handled by storing multiple API contexts on a stack, where the topmost context on the stack is used by the most recent API routine in the call stack.

  Nodes on the API context stack are represented by the struct `H5CX_node_t`. Each node contains a pointer to an `H5CX_t` instance with the actual information for that context, and a pointer to the next (lower) node in the stack.

Context Field Type	Description
Reference Property Lists	The property lists to use for this operation. It may be default or provided by the application. The context stores pointers to the underlying property list objects (`H5P_genplist_t`) without copying them. These fields retrieve property lists for operations that require them in modules such as `H5FD`.
Internal Fields	Used for managing state that doesn't correspond to a property from a property list. Generally set at a high level and used at a low level. For example, the metadata "tag" is set in high-level group/dataset code and used in the low-level metadata cache.
Cached Fields	Cached fields correspond to properties from property lists. In general, context fields are retrieved from the underlying property list and saved in the context on-demand. However, if the context contains a default property list, the context fields are retrieved from cached fields instead. These cached fields are populated from the default property list during library initialization. These cached accesses are faster due to bypassing property list operations. Changes to cached field values do not propagate to the original property lists.
Return-Only Fields	Return-only fields correspond to properties from property lists that are never retrieved within the library and are only set with values in order to return those values to the application. When a value is set in a return-only field, that value is cached in the context until the context is destroyed when the API routine returns. At that time, any changed return-only fields have their new values written into the provided property list(s).

Table 4.1. – API Context Field Types

Each API call uses the macro `FUNC_ENTER_API` (or `FUNC_ENTER_API_NOCLEAR`) to initialize and allocate a new context node. This context node is pushed to the top of the context stack by `H5CX_push()`. The end of each API call uses the macro `FUNC_LEAVE_API` to free the context node and remove it from the context stack with `H5CX_pop()`.

`H5CX_push_special()`/`H5CX_pop_special()` are only used in the library termination routine `H5_term_library()`. The normal push and pop routines use the library's free-list memory allocation and free routines. During library termination, the free-list and memory management structures are shut down. Thus, to avoid dependency on these structures, the special push and pop functions use system memory calls instead.

- API Context Setup

  API routines need to use certain functions, depending on their parameters, to correctly set up their API context. The use case and description for each of these functions are in Table 4.2.

- Thread Safety and Parallel

  Each API context stack is thread-local. As a consequence, API contexts are inherently thread-safe, and API contexts do not require special consideration when using parallel HDF5. Several fields in the

Context Function	Use Case	Description
`H5CX_set_apl()`	Routines with an access property list parameter	Stores the access property list in the context, sets the collective metadata read flag in the context, and enables sanity checks during collective API operations.
`H5CX_set_loc()`	Routines that modify file metadata **without** an access list parameter	Sets the collective metadata read flag in the context, and enables sanity checks during collective API operations.
`H5CX_set_lapl()`	Routines which have both an object-specific access property list, and a link access property list (LAPL)	When a routine takes two access property lists, the object-specific access property list should be set with `H5CX_set_apl()`, and the LAPL should be set with `H5CX_set_lapl()`.
`H5CX_set_dxpl()`	Routines with a data access property list (DXPL) parameter	Stores the provided DXPL in the context.

Table 4.2. – API Context Setup Routines

context are used to store and retrieve information about collective operations for the MPI-IO VFD, but this procedure is rank-local and entirely adheres to the usage pattern of other context fields.

4.1.4. Error Handling (`H5E`)

- Error Messages

  An error message is an in-memory object represented by the `H5E_msg_t` struct. An error message contains a string describing the error that occurred, a 'type' indicating whether the error is major or minor, and a pointer to the error class it belongs to. Because error messages contain a dynamically allocated error string, they are reference-counted objects that are assigned handle IDs by the library, and have their own create and close API (`H5Ecreate_msg()`, `H5Eclose_msg()`).

- Error Records

  Error records are complete descriptions of individual errors. An error record is implemented as the `H5E_error2_t` struct, which consists of a major error message, a minor error message, an error class, a description of the specific error that occurred, and the location at which the error occurred in the source code.

- Error Stacks

  An error stack is a collection of error records. By default, when an error occurs in a library function, an error record is pushed onto the active error stack and that function returns a failure indication. Its caller detects the failure, pushes another error record onto the stack, and returns another failure indication. This continues until the API function called by the application returns a failure indication. The library has a default error stack. Generally, errors cause this stack to be automatically printed to the standard error stream. `H5E_DEFAULT` may be provided as an error stack ID to indicate the library's default

error stack to error API functions. The library's currently active error stack is stored as the globally accessible H5E_stack_g.

An error stack is implemented as an H5E_t struct. Each error stack is created with space for a fixed amount of error records. If more than this amount of errors are pushed onto a single stack, excess error records are truncated, and only the innermost errors will remain.

When an error record is pushed onto an error stack, the information of the record is not copied. Pointers to the record's error class and error messages are stored, and H5I_inc_ref() increases the reference count of those objects. Correspondingly, when a record is removed from a stack, these pointers are removed, and the reference count to the record's fields is decreased via H5I_dec_ref().

Most top-level API functions clear the active error stack upon entry through the FUNC_ENTER_API macro, which calls H5E_clear_stack(). Certain error-handling API functions, such as H5Eprint(), do not clear the error stack since this would destroy the information they are meant to work with. Error clearing is avoided by replacing the API entry macro with the similar FUNC_ENTER_API_NOCLEAR. Private library functions, which call FUNC_ENTER_NOAPI upon entry, and package library functions, which use FUNC_ENTER_PACKAGE upon entry, do not clear the error stack.

- Error Reporting Macros

  The macro FUNC_GOTO_ERROR usually handles error reporting in library functions. This macro takes a major and minor error code, a return value to indicate failure and a message. If automatic error reporting is enabled, it uses the preprocessor macros __FILE__, __func__, and __LINE__ to add an error to the active error stack with information about the file, function, and line on which the error occurred. Afterward, it jumps to the done label of the parent function, where cleanup and failure handling are generally performed.

  If an error occurs after the done label, the use of FUNC_GOTO_ERROR could potentially result in an infinite loop. To prevent this, the macro FUNC_DONE_ERROR performs the same error reporting without jumping to the done label and is used whenever an error occurs during function cleanup or error handling.

- Error Classes

  An error class is a collection of major and minor error messages. Error records must belong to an error class. An application can register a new custom error class with the library via H5Eregister_class(). An error class is implemented as a H5E_cls_t struct containing only the three strings provided to H5Eregister_class(), consisting of the error class's name, the library to which the error class belongs, and the version of that library. When using H5Eprint(), the error class of a set of records is displayed as <ERROR_CLASS>-DIAG.

- Customized Error Handling

  Users can define their own error reporting callback of type H5E_auto2_t. When an API function returns a failure indication, this callback will be invoked on the error stack. The default error reporting callback, H5Eprint(), uses H5Ewalk() to traverse the individual error records within an error stack. H5Ewalk() invokes a H5E_walk_t callback on each error record and traverses the error stack.

  H5Eget_current_stack() registers the library's current error stack as an in-memory object and assigns it a handle ID. This clears the active error stack but leaves the error records that were in the

active error stack accessible within the returned error stack object. Once an error stack is assigned a handle, custom error handling can be performed directly through the error API.

4.1.5. Library Identifiers (H5I)

HDF5 IDs are library-managed, reference-counted handles returned to the user when they create or open HDF5 files, datasets, etc. This allows the library to handle state for the user, who only needs to keep track of the returned identifier and not a pointer to an object's memory, which would be more error-prone.

hid_t Values

HDF5 IDs are represented using the hid_t type, which is a 64-bit signed integer. IDs are broken down into "ID types" (H5I_type_t) such as H5I_DATASET and H5I_VOL that are managed as separate groups. The first TYPE_BITS bits of the hid_t value represent this type, with the remaining bits representing the ID's value. This is currently 7, which allows 128 ID types (the library currently uses < 20, allowing users to define ∼100) and each ID type can represent $2^{64-7} = 144,115,188,075,855,872$ IDs. The special H5I_INVALID_HID value is defined to -1 and is used as an error hid_t value.

hid_t IDs are a transient library state management system and are not a part of the HDF5 file, so, for example, an HDF5 file will not always have the same ID if it's opened under different circumstances. In fact, an HDF5 object can be concurrently opened multiple times, returning different IDs each time. This is usually handled via a library object type scheme where object data are split between H5X_t and H5X_shared_t structs, with per-hid_t data being stored in the H5X_t struct and everything else going to H5X_shared_t, which is accessed via a pointer in H5X_t.

In general, hid_t values are for use by users, not the library. Upon entry to the library, hid_t values are almost always immediately unwrapped as described below. It is rare that hid_t values are passed around inside of the library. One exception to this is refresh API calls, where we close the underlying object and reopen it. In that case, we have to reattach the re-opened data to the original ID.

ID values are not recycled, so a (very) long-running process could run out of them.

Storing and Accessing ID-Managed Data

Each ID type stores its objects in a hash table. The stored ID data include its hid_t value, a pointer to the stored object, and some metadata like the reference count of the object. At library initialization time, the has tables are constructed and associated with other ID type info, such as the type's free function.

When an ID is requested for an object, the object/ID/metadata are simply stored in the hash table with a reference count of 1.

When a hid_t value is passed into the library, it's typically immediately unwrapped via H5VL_object() (VOL-related objects) or H5I_object() (everything else) to get at the underlying data. These calls do not remove the object from the hash table, but simply get a pointer to the stored object. Since the data pointed to are managed by the H5I code, nothing special has to be done in the library when the pointers are no longer needed.

When an ID is no longer needed and closed, the reference count is decremented. If it's the last reference to an object, the ID type's free callback is invoked, fully closing the object and freeing resources.

At library shutdown time, all ID types are cleared and any still-open objects will be closed.

Extensibility

Users can define their own ID type using the `H5Iregister_type()` API call. IDs for user-created objects can then be obtained using `H5Iregister()`. There are many other `H5I` API calls that can be used to manipulate user-defined IDs and ID types. Most of these API calls can ONLY be used on user-defined types. For example, you can't call `H5Idestroy_type()` on a library ID type as those are managed internally and shut down when the library shuts down. User-defined types are otherwise treated identically to library-defined types inside the library.

Special Considerations

Datatypes Datatypes can be both committed and non-committed, with the former requiring the same object-handling mechanism as other VOL objects. This is currently handled via special datatype code paths that use code in the `H5T` package to check if the stored object is a committed datatype or not and then manage the data appropriately.

Deletion When IDs are deleted in `H5I_clear_type()`, the IDs can't simply be deleted during the traversal, as the type's `free` callback could free other IDs, potentially corrupting the internal data structures. Instead, this is implemented using a "mark-and-sweep" algorithm to avoid problems.

4.1.6. Metadata Cache (`H5[AC,C]`)

The metadata cache is a cache meant to speed up access to HDF5 metadata and supports features such as cache size adjustment under automatic or user program control.

The cache itself is like a very modified version of the UNIX buffer cache and uses a "Least Recently Used" (LRU) replacement policy to determine which items within the cache get evicted first. Due to the fact that the metadata written to a file can grow without bounds, the size of the metadata cache must reflect this ability, and no max entry amount can be set. Instead, users may set a maximum size, and eviction will occur as necessary using the LRU replacement policy. Similarly to UNIX, when dirty entries reach the end of the cache, they are flushed, marked as clean, and reset to the beginning of the cache.

When running in parallel, there are three different strategies for writing metadata to the file. By default, all pieces of metadata are distributed among the ranks, with each rank writing its pieces independently and marking the others clean. Optionally, metadata can be distributed among the processes in the same way and then written in a single collective operation. Finally, the library can be configured so that only process 0 is allowed to write metadata to the file, and it does so only at synchronization points.

It is possible to tell when the size of the cache is a problem and must increase based on the hit rate, which is calculated over a period of time known as an epoch. When the hit rate has been low for a while, it means that

the cache is probably at maximum capacity, and its capacity is perhaps not big enough. If this happens, the size will increase according to some user-defined multiple.

In addition, if the metadata entry is larger than the cache itself, this could also cause problems. This issue is fixed whenever it appears by taking the size of the metadata entry, multiplying it by some preset factor, and increasing the size of the cache based on this new size.

Both of the above problems may arise when cache sizes are too big. Similarly, issues can occur when the cache size is too small. However, this is harder to judge. Ageout with hit rate threshold is the default cache size reduction algorithm of choice, and it combines two methods of detecting an oversized cache size. The first involves checking the hit rate once again. If the hit rate exceeds some preset amount, it might mean that the cache size is too big and needs to be reduced. Then, the second part of the algorithm, checking for metadata entries that haven't been accessed in some number of epochs, kicks in and removes aged-out entries.

The metadata cache's configuration information is stored in a `H5AC_cache_config_t` struct. This struct contains fields for general information, size-increasing control fields, size-decreasing control fields, and parallel configuration fields.

For a more complete reference of the metadata cache, please refer to https://docs.hdfgroup.org/hdf5/develop/_t_n_m_d_c.html or the source code.

4.1.7. Files and the Open File List (`H5F`)

- File in-memory objects

  Each successful `H5Fopen()` results in a top-level file descriptor `H5F_t`. By contrast, `H5F_shared_t`, the underlying structure storing information about the opened file, is allocated only once for each file in storage and is shared between file descriptors that point at the same file. Most of the fields in `H5F_shared_t` represent information related to persisting HDF5 files in a file system according to the file format specification [10]. The shared file also tracks how many file descriptors point to it and is only closed when its last file descriptor is closed.

  The shared file contains a list of all opened objects, while each file descriptor maintains only counts of how many times each object is opened through that particular file descriptor. This division of information ensures that closing a file descriptor only decrements references that originated through that descriptor. For similar reasons, each file descriptor tracks the total number of objects it has open, excluding the root group.

  Most information about child files mounted onto a parent file is kept in a table on the shared file structure of the parent file. The file descriptor maintains only a count of how many files were mounted through itself, similar to its count for open objects. The number of mounted files is maintained so that a file descriptor will close when 1) a weak close (see the section 'File open and close') was previously performed on it, and 2) an object is closed, and the only open objects remaining on the file descriptor are mounted groups.

  The shared file also contains information necessary for I/O, such as the file's permissions in storage, a page buffer, and any free space managers.

- Open file list

The file module maintains a list of open shared file objects, `H5F_sfile_head_g`. Newly opened files are appended to the head of this list. This list is searched when checking if a file is already open. The actual comparison to determine if a potentially non-open file matches an open file is handled by the active VFD, which is not required to provide a way to compare files. If it does not provide such a method, then the application opening multiple handles to the same file will result in undefined behavior, and it is the application's responsibility to ensure that the same file is not opened multiple times.

The open file list is implemented as a linked list. Due to the need to search this list upon each file open, performance may degrade if there is a large number of open files. The library is optimized for a small number of top-level files that contain many groups and objects in their internal file hierarchy.

- File open and close

  File opening is first attempted without creating or truncating any files. This allows attempts to truncate or re-create currently opened files to safely fail without modifying any state. If the file is not already open, then the open proceeds normally, modifying files as requested, and the resulting shared file object is added to the open file list.

  Each FAPL has a 'file close degree', which determines how open objects within a file are handled at file close. The 'weak' close degree allows the file handle to be closed from the application's perspective (as tracked by the `closed` flag on a file descriptor), but keeps the file handle around until all objects under it are closed. The root group and the root groups of any mounted files are not considered open objects for this purpose, and a weak close will be completed if such groups are the only open objects remaining. The 'semi' close degree causes a file close to fail if any objects remain open. The 'strong' close degree causes any open objects to be closed before the close completes. Most VFDs default to 'weak', except for the MPI-I/O driver which defaults to 'semi', due to file closes being a collective operation. When HDF5 files are mounted, the parent and child files must have the same close degree in order to avoid contradictions in how to handle objects at close time.

  Closing an object may or may not evict it from the file's metadata cache, depending on the 'evict on close' setting. This setting trades increased memory with a larger cache for higher speed, or lower speed with a smaller cache for decreased memory usage.

- External file cache

  The external file cache (EFC) is a mechanism for keeping files opened through external links quickly accessible for subsequent accesses. Each shared file structure may have an EFC that stores information about files accessed through external links in that shared file.

  EFC entries are linked to the top-level file descriptor, as the EFC caches the result of opening an external file (a top-level file descriptor). Because file descriptors are not reference counted, EFC entries must be reference counted within the EFC in order to handle dependencies in the EFC tree.

  A file's EFC contains both a skip list of cache entries and the head of a cache entry list sorted by the least recent update (LRU) time. The skip list is used to check whether a newly opened external file is already present in the cache. The LRU-sorted list speeds up the common case of closing the most recently opened external file and allows the least-recently accessed cache entry to be dropped when the maximum size of the EFC is reached.

4.1.8. File Page Buffer (H5PB)

The page buffer exists within the native VOL, just above the VFL and just below the I/O layer. Thus, the library can benefit from the page buffer cache regardless of which VFD is active.

There are a few conditions under which file I/O bypasses the page buffer entirely: if page buffering is disabled, if the requested I/O operation is larger than one page and thus would not benefit from the page buffer, or if the requested operation is a parallel raw data access. Additionally, when opening a file with page buffering enabled, the superblock contains information needed to initialize the page buffer (e.g. page size). Because this information must be read from the superblock before the page buffer is initialized, the page buffer cannot be used for this operation.

The page buffer's page size will be rounded down to the nearest multiple of the file space page size, in order to avoid situations where reading or writing in blocks of page size could extend outside the bounds of the file.

- In-memory objects

  A page buffer is represented in memory by the struct `H5PB_t`, which is attached to a shared file object. Page buffers are associated with the shared file so that different handles to the same file share the same page buffer.

  The page buffer maintains three lists of pages. The first is a skip list of page entries allocated by the page buffer itself. The second is another skip list containing only the page entries inserted by the free space manager. The free space manager inserts blank pages into this list during reads of file regions that have never been written to, in order to avoid unnecessary file accesses. The third list is a linked list of page entries sorted by the least recent update (LRU). The LRU-sorted list is used to evict the oldest page entries when the maximum size of the page buffer is reached.

  Page buffer entries (`H5PB_entry_t`) consist of their location in the file, pointers used to maintain the LRU-list, and an indication of whether the page buffer entry contains metadata or raw data.

- Metadata vs Raw Data

  I/O operations accessing a metadata entry are always atomic - the entire entry is always accessed. Noncontiguous subsets of raw data may be accessed using selections, and so I/O operations on raw data through the page buffer require some additional steps. During raw data writes, the page buffer needs to update any pages that are partially modified by large (greater than page size) writes, and discard any pages that are entirely overwritten. Similarly, during raw data reads, the page buffer must collect data from any dirty pages that are within the span of the I/O request.

  The utility of storing metadata versus raw data in the cache will vary from use case to use case. To allow applications to optimize for their particular case, and to avoid a scenario where one type of data is completely evicted from the page buffer cache, applications may set a minimum percentage of the page buffer's memory to be reserved for each type of data.

  A page in the page buffer must contain only metadata or only raw data. This restriction is imposed in order to properly map pages to regions of the file in storage, as metadata and raw data are not interleaved within pages in storage (with the default driver).

- Page Buffering in Parallel

  Parallel HDF5 has limited support for the page buffer due to the need for all processes to have a mutually consistent view of the data.

Because all operations that modify metadata must be performed collectively, the page buffer can be used during parallel metadata I/O if the metadata write strategy has a single process handle all writing of dirty metadata from cache. This is accomplished by having the writer-process write all dirty entries to the page buffer, flush the page buffer to storage, and then update the other ranks with the metadata entries that were written.

The page buffer is always bypassed when reading or writing raw data in parallel. When accessing raw data collectively, the HDF5 library constructs MPI-derived datatypes to represent possibly non-contiguous buffers in memory and file offsets. At the page buffer layer, it would be necessary to flatten the derived datatypes, update the page buffers, and then impose additional communication among all processes to determine who reads/writes which pages for the collective operation. This would represent considerable overhead and complexity, and replicate work that is more logically handled by MPI itself. Simply bypassing the page buffer likely leads to better performance than this implementation would. Additionally, most parallel raw data accesses are large enough that the page buffer would not provide a significant benefit. While many of these issues would not apply to parallel independent raw data access, concurrent use of collective and independent operations would cause the page buffers of the processes involved in each operation to become inconsistent with one another, and thus the page buffer is bypassed during all parallel operations regardless of whether they are collective or independent.

When using Single Writer Multiple Reader (SWMR), the writer may not make use of a page buffer. The SWMR writer has a required order for its operations, which would not be preserved by the page buffer. SWMR readers can use the page buffer with only one caveat. After a refresh operation is performed on an object, entries that get evicted from the metadata cache must have their corresponding pages evicted from the page buffer in order to avoid subsequent reads returning old data.

4.1.9. File Objects (`H5[O,SM]`)

File objects are the heart of the HDF5 data model. In the library, the object package (`H5O`) consists of structures and code common to all object types, as we use code to handle object headers, which are the central on-disk objects for storing HDF5 file objects. All objects contain an object header, which contains some fixed metadata and a variable list of object header messages, as well as any data referenced by those object header messages.

- In-memory objects

  The central in-memory structure for an object is `H5O_t`, which is defined in `H5Opkg.h`. This structure essentially contains the information that is stored in the object header, including the list of object header messages. While it might seem natural for the type-specific object structs (such as `H5D_t` for datasets) to contain an `H5O_t` at the top (beginning) of the struct, this is not the case. `H5O_t` is essentially a lower-level proxy for the object header on disk, while the type specific structures are closely tied to the logical objects exposed through the API.

  One potential point of confusion is the `rc` field in the `H5O_t` struct is not the same as the `rc` field in the struct returned by `H5Oget_info()`. The `rc` in `H5O_t` is the in-memory reference count of the memory struct itself, while the `rc` returned by `H5Oget_info()` is the number of links to the object header in the file. The `nlinks` field in `H5O_t` is equivalent to the link-counting `rc` from `H5Oget_info()`.

- Object headers

The object header is the root piece of metadata for all HDF5 objects on disk. The object header consists of the main object header block, which in turn consists of the prefix containing generic object metadata, as well as some number of object header messages. If the number of messages grows too large to fit into the main object header block, one or more object header continuation chunks will be created on disk, each containing more object header messages. Object header continuation blocks are pointed to from their parent object header using an object header continuation message.

Since an object header is not a single block on disk, but is treated as a single object by most of the library, the way the metadata cache handles it is slightly strange. There are metadata cache clients for object header root blocks and continuation blocks, each of which serializes and deserializes these objects as needed. However, when the entire object header is retrieved, typically using `H5O_protect()` or `H5O_pin()`, which take an `H5O_loc_t` object location, these functions use the metadata cache to decode all the constituent blocks of the object header, protects (and optionally pins) the root object header block, then assembles and returns an `H5O_t`. The `H5O_loc_t` struct is how other packages generally keep track of their objects and mostly consists only of a file and address of the root object header block.

- Object header messages

  Object header messages encode the majority of the important metadata of the object. There are several different types of object header messages, and only some of them are present in any one object header. The type of an object is determined by which object header messages are present. For example, if a layout message is present, then the object is a dataset. Object header messages also appear in the file superblock, even though that is not an object header.

  Each object header message type is defined in a separate source file, for example, `H5Olayout.c` for layout messages. Each message type defines itself using the `H5O_msg_class_t` struct, which contains callback functions that must be implemented (some are optional) and other basic information about the message type. Most object header message types can only be present once in an object header, but there are some exceptions, such as attributes, link messages, and object header continuation messages. This distinction is important since `H5O_msg_read()` only reads the first message of a given type, so if there may be more than one message of that type, `H5O_msg_iterate()` must be used instead, with the callback operation checking to see if it is the correct message. The generic (type agnostic) object header message code stores messages in the `H5O_mesg_t` struct. This struct contains information on allocation within the object header, as well as pointers to the serialized/on-disk form (`raw`) and the deserialized/in-memory form (`native`). The in-memory structs for all the different message types are defined in `H5Oprivate.h`. They are defined in the private header because other packages widely use the native forms of these messages. This makes `H5Oprivate.h` one of the most important header files in HDF5, and can be thought of as a significant contributor to the coupling between the packages.

  Allocation of space for object header messages is somewhat complicated and handled in `H5Oalloc.c`. This code handles allocation, deletion, free space tracking (with null messages), and the creation of new continuation blocks (in conjunction with the metadata cache and standard file space allocation code). This code needs to handle several corner cases, such as when a message is added when there isn't enough space for a continuation message (in this case it must move messages), or if there's empty space but there isn't space for a null message (in this case it encodes a "gap" value in the object header).

- Shared object header messages

To save space, the HDF5 library can optionally automatically check if object header messages are the same between different objects, and only store this message once if that is the case. Such messages are called shared object header messages, and can be stored either in one object's object header, or in the file's shared object header message heap. Additionally, these shared messages can be indexed either with a simple list or a v2 b-tree. Using a v2 b-tree obviously improves the performance of checking to see if a message is already shared, but requires a more recent file format version.

Object header messages that can be shared need to follow special steps in their definition files. First, they need to `#include H5Oshared.h`, then, for the callbacks which are defined in `H5Oshared.h`, they need to `#define` the callback from `H5Oshared.h` to a name specific to the message type (this name isn't actually defined in the message file, it's used to point to the shared function from the header), then do the same with the symbol that ends in `_REAL`, defining it to be the name of the actual function implemented in the object header message definition file. For example, `H5Odtype.c` uses: `#define H5O_SHARED_DECODE H5O__dtype_shared_decode`, where `H5O__dtype_shared_decode` is only used as the name for the `decode` callback in the `H5O_msg_class_t` struct; `#define H5O_SHARED_DECODE_REAL H5O__dtype_decode`, where `H5O__dtype_decode()` is defined as the actual datatype decoding function in `H5Odtype.c`, which does not handle shared messages. Therefore, when the library decodes a datatype message, the code flows into `H5O_SHARED_DECODE()`, which locates the actual datatype message and then invokes `H5O__dtype_decode()` to do the actual decoding.

For callbacks that are not implemented by the object header message definition file, they must still `#define` the callback from `H5Oshared.h` to a name and include that name in the class struct, but they may then `#undef` the `_REAL` version of the callback. In addition, the object header message definition file must `#define H5O_SHARED_TYPE` to the name of the global variable containing its `H5O_msg_class_t` class struct.

Additional code for handling shared messages in these callbacks is defined in `H5Oshared.c`, while most of the code for handling shared messages (allocation, search, etc.) is contained in the `SM` package. Code for handling the message in the file superblock which points to the shared message heap is contained in `H5Oshmesg.c`.

4.1.10. Attributes (`H5A`)

- Attribute in-memory objects

  Attribute in-memory objects are represented with the `H5A_t` structure, defined in `H5Apkg.h`. This structure contains information about whether the attribute is shared between multiple objects, the location of the attribute's parent object on disk, and the path to the attribute in the file hierarchy. The underlying shared attribute object `H5A_shared_t` contains information such as the size of the attribute, its dataspace and datatype, and its name. The shared `nrefs` field tracks the number of remaining references to the attribute from other objects in the file, and the attribute's resources are freed once this count is decremented to zero.

  The `crt_idx` field tracks the creation index of the attribute, if creation index tracking is enabled on the parent object. This index only tracks the order in which attributes were created relative to each other, not any absolute information about the time of their creation.

- Compact and Dense Attribute Storage

There are two kinds of attribute storage: compact and dense. Compact attributes are kept in the object header of the parent object, along with other metadata for that object. Dense attributes are kept in a fractal heap. The address of the fractal heap for dense attribute storage is kept in the Attribute Info message in the object header. The heap IDs to access specific dense attributes in the fractal heap are stored in a name-index B-tree.

If the file access property list used for a file is set for file format compatibility with library versions before 1.8 (as is the case by default), all attributes will be stored compactly, and the creation of attributes larger than 64 KiB will not be allowed.

The architecture for attribute storage is nearly identical to the architecture for link storage.

- Attribute object header messages (open/flush)

  The primary fields that define an attribute are its datatype, dataspace, name, and the data it contains. When an attribute is compact, all of these fields are stored in an attribute object header message and inserted into the object header of the parent object. In library file format versions 1.6 and later, the datatype/dataspace fields in an attribute message may be pointers to a datatype or dataspace message that is shared between objects.

  If an object is using dense attribute storage, then the object header will contain an attribute info message with the location of a name-index B-tree, as well as the address of the fractal heap storing the attribute messages.

  Changes to object headers make use of caching for optimization. Attribute messages may be flushed to storage by use of `H5Fflush()` on the containing file, or `H5(D/G/T/O)flush()` on an object of the appropriate type with attached attributes. Note that these functions only directly control the library's buffers, not any buffering performed by the operating system. Flushing of object headers is also performed automatically by the library during certain operations.

- Attribute I/O

  Attributes do not support partial reads or writes using selections. Any read or write to an attribute will operate on the attribute's entire dataspace. Attributes also do not support compression or chunking. Because attributes are intended to hold small quantities of metadata, these aspects of the I/O pipeline were omitted in order to optimize and simplify attribute I/O. If any of these I/O features are desired for use with metadata, then that metadata may be stored in a dataset instead, and a pointer to that dataset stored in an attribute - see the section "Dataset Storage Layouts".

  Reads and writes to attributes use the same datatype conversion pipeline as read and writes to datasets.

  If an attribute in compact storage is written to, all of its previous data in the object header is overwritten by an in-memory buffer that is later flushed to storage. If an attribute in dense storage is written to, a write operation is performed on the fractal heap at the heap ID storing the attribute by name.

- Attribute iteration

  The iteration process varies between compact and dense attributes.

  Dense attribute iteration is handled by `H5A__dense_iterate()`. If the iteration order was specified to be `H5_ITER_NATIVE` in order to optimize for speed, then the order is determined by the hashes of the attribute names in the name index B-tree. Otherwise, a table of attributes is built by `H5A__dense_build_table()`, sorted in the desired order by `H5A__attr_sort_table()`, and then iterated through by `H5A__attr_iterate_table()`.

The compact iteration case is similar: H5A__compact_build_table() creates a table of attributes, and iterates through it via H5A__attr_iterate_table(). For compact attributes, the table is always constructed, even when the iteration order is H5_ITER_NATIVE.

The attribute table itself is the same, regardless of the storage type its attributes were retrieved from.

The architecture for attribute iteration is nearly identical to the architecture for link iteration. As with all forms of iteration in the library, attributes should not be created or removed during iteration - changes to the parent object's attribute count during iteration can lead to undefined behavior.

4.1.11. Groups and Links (H5[G,L])

A group is a potentially empty collection of link objects. The name of an object within a group is actually the name of a link that refers to it. The same underlying object may be referred to by multiple names, if it is pointed to by multiple links in one or more groups. When an object is created by a user, the library automatically creates a new hard link from the parent group to that object.

- Group in-memory objects

  Groups are represented by the H5G_t struct. This struct has three elements: the location of the group within the file hierarchy as an H5G_name_t instance, the group's address within the file as an H5O_loc_t instance, and shared information for the group as an H5G_shared_t instance. The shared group information is available to all in-memory objects which refer to the same underlying group in the file, and allows a shared count of how many in-memory objects reference that particular group to be maintained.

- Link storage

  There are two kinds of link storage: compact and dense. Compactly stored links are kept in the object header for the group containing them, along with other metadata for that group object. Densely stored links are kept in a fractal heap. The address of the dense link fractal heap is then stored in the Link Info message in the group's header, with the heap IDs to access specific dense links stored in the name index B-tree.

  If the file access property list used for a file is set for file format compatibility with library versions before 1.8 (as is the case by default), all links will be stored in the file's symbol table.

  The architecture for dense/compact link storage is nearly identical to the architecture for attribute storage.

- User-defined links

  Users may register a user-defined link class with H5Lregister(), and then create a link of that class using H5Lcreate_ud(). A link class definition must at minimum include a version number for the link class struct H5L_class_t, a link class identifier class_id, and a traversal function trav_func. link_class must be in the range between H5L_TYPE_UD_MIN and H5L_TYPE_UD_MAX. Note that external links are implemented as an example of a user-defined link class, using class_id = H5L_TYPE_UD_MIN.

  User-defined link classes may also provide optional callback functions to create, move, copy, delete, and query an instance of the link class. Some functionality is automatically implemented for these callbacks as a consequence of the type-independent link architecture.

Depending on the user's implementation of the link callbacks, user-defined links may or may not affect the `nlinks` count of objects, and may or may not be allowed to dangle.

- Link traversal

  "Link traversal" refers to the process of following links to their target objects. When opening an object through a link, the link's target location in the file hierarchy is iteratively determined by `H5G_traverse()`, and once the target location is reached, location information specifying position within the file hierarchy (`H5G_name_t`) and within the file (`H5O_loc_t`) is copied to the opened object.

  For hard links, if the hard link points to a mounted file, then `H5F_traverse_mount()` performs a binary search over the mount table of the original file in order to copy the root group location information of the mounted file.

  For soft links, traversal of the target path is performed by `H5G__traverse_slink()`. The file hierarchy traversal process of `H5G_traverse()` is repeated on the path specified by the soft link. Because soft links are allowed to dangle, this traversal may find that no object exists at the target location, or that the object is of an unexpected type.

  For user-defined links, the provided traversal callback is invoked in `H5G__traverse_ud()`.

  For external links, the traverse callback `H5L__extern_traverse()` handles opening the external file and locating the target object within it.

- Link iteration

  Iteration over all links in a group may be performed through `H5Literate()` non-recursively, or `H5Lvisit()` recursively. `H5Literate()`, when using the native VOL, passes control to `H5L_iterate()`, which internally uses `H5G_iterate()`. Similarly, `H5Lvisit()` ultimately passes control to `H5G_visit()`. Making the relationship between groups and links clear was one of the reasons that the iteration functions in the `H5G` module were deprecated in favor of `H5Lvisit()/H5Literate()`.

  The iteration process varies based on whether the group is currently using dense or compact storage (or a symbol table, for old versions of the file format).

  Dense link iteration is handled by `H5G__dense_iterate()`. If the iteration order was specified to be `H5_ITER_NATIVE` in order to optimize for speed, then the order of link iteration is determined by the hashes of the link names in the name index B-tree. Otherwise, a table of links is built by `H5G__dense_build_table()`, sorted in the desired order by `H5G__link_sort_table()`, and then the link table is iterated through by `H5G__link_iterate_table()`.

  The compact iteration case is similar: `H5G__compact_iterate()` uses `H5G__compact_build_table()` to create a table of links and iterates through it via `H5G__link_iterate_table()`. For compact links, the link table is always constructed, even when the iteration order is `H5_ITER_NATIVE`.

  The link table itself is the same, regardless of the storage type its links were retrieved from.

  The architecture for attribute iteration is nearly identical to the architecture for link iteration.

4.1.12. Datasets (H5D)

- Dataset objects

  A dataset is an HDF5 object which stores raw array data. Each dataset must include layout, dataspace, and datatype object header messages, and may optionally include pipeline, fill value, and EFL (external file list) messages in addition to generic messages like attributes and modification time. The layout message stores information about how the data is laid out in the file, for example, the location of the address for the chunk index, the location of the contiguous data block, or the actual data for compact datasets. The dataspace message contains the extent (dimensions) of the dataset. The datatype message contains a description of the format of a single element of data in the dataset as stored in the file.

- Dataset in-memory objects

  Like other HDF5 objects, a dataset object in memory consists of a top-level struct, `H5D_t`, which maps one-to-one with IDs, and a shared struct, `H5D_shared_t`, which can be used by multiple `H5D_t`s. The `H5D_t` consists only of an `H5O_loc_t` object location, an `H5G_name_t` path name, and a pointer to the shared struct. The `H5D_shared_t` shared struct contains the layout, dataspace, and datatype information as expected, as well as the dataset creation property list (DCPL) and dataset access property list (DAPL), cached information from the DCPL and dataspace, information on the raw data cache (if any), and a few other pieces of information used in a transient manner. The dataspace, datatype, DCPL, and DAPL are referenced by an `hid_t` ID, while the layout is stored in the struct as an `H5O_layout_t` layout message struct.

- Contiguous datasets and the sieve buffer

  Contiguous storage is the default storage type for HDF5 datasets. A contiguous dataset contains a single data block in the file, with its address stored in the layout message in the dataset's object header. Contiguous datasets cannot be resized, and do not support data filters. Similarly, in memory, the struct `H5O_storage_contig_t` contained in `H5O_layout_t` contains only the address and size of the contiguous data block.

  Contiguous datasets are conceptually very simple, but they do support a form of caching with the sieve buffer. The sieve buffer consists of a single block of cached data that is contiguous in the serialized (flattened) dataset. The maximum size of the sieve buffer is configurable and can be disabled by the file driver if it could cause problems (such as for the MPIO file driver). The sieve buffer exists at a specific offset in the dataset's data block, and I/O within that block is performed to/from the buffer. I/O outside of the sieve buffer causes the existing buffer to be flushed and then moved to the location of the new I/O and reconstituted from disk (unless it is being fully written). I/O that exactly adjoins the sieve buffer boundary (before or after) will extend the sieve buffer without flushing it unless doing so would cause the sieve buffer to be too large. The sieve buffer is a very low-level construct - it has no knowledge of the dataspace or dataset, and only operates on a one-dimensional block of bytes.

  In addition to being used for contiguous datasets, many of the I/O routines in `H5Dcontig.c` are used in other places to perform I/O on a single contiguous block of array data on disk. This is covered in greater detail in the section on the I/O pipeline.

- External datasets

  External datasets are similar to contiguous datasets except their data is stored in an external file instead of a contiguous block in the HDF5 file. They are internally treated as a special case of contiguous

datasets, and similarly cannot be resized and do not support data filters. The layout type in the datasets' `H5O_layout_t` layout field (and that in the layout message in the file) is `H5D_CONTIGUOUS`, but when the number of external files in the external file list message (the `nused` field in the `H5O_efl_t`, stored in the dataset's DCPL cache) is non-zero, the dataset is using external storage, with the associated layout operations.

External datasets reuse the contiguous layout's `ser_read` and `ser_write` routines, which simply pass the call down to the lower level, but use their own low-level `readvv` and `writevv` callbacks, which are called after unpacking the selections and performing any type conversions, and which perform the actual interfacing with the external files.

- Compact datasets

  Compact datasets store all the raw data directly inside the dataset's object header, similar to attributes. In this case, the layout message contains the raw data block, and similarly, the data buffer in memory is contained in the `storage` field of the `H5O_layout_t` struct, contained in the main `H5D_shared_t` struct. Like external datasets, compact datasets reuse the contiguous layout's `ser_read` and `ser_write` routines, which simply pass the call down to the lower level but use their own low-level `readvv` and `writevv` callbacks, which are called after unpacking the selections and performing any type conversions. These compact callbacks perform in-memory I/O to the data buffer in the layout struct. Actual I/O to the file occurs when the dataset's object header is flushed or loaded by the metadata cache. Compact datasets do not support compression or data filters.

- Chunked datasets and the chunk cache

  Chunked datasets break their raw data arrays into constant-sized, tiled, N-dimensional blocks (rectangles in 2-D datasets), and store each of these blocks (called chunks) as a single contiguous block in the file. A contiguous dataset is conceptually similar to a chunked dataset with the chunk dimensions set to be equal to the dataset dimensions, though in practice these two cases are different in several ways.

  Since chunks are located separately on disk, there needs to be a way to look up the index of each chunk. This is done via the chunk index, of which there are several types. These index types are described in more detail in the relevant sections of this document, but it is important to cover how the dataset chunk code uses these indices. The chunk indexes are defined in the files `H5Dbtree.c`, `H5Dbtree2.c`, `H5Dfarray.c`, `H5Dearray.c`, `H5Dsingle.c`, and `H5Dnone.c`. With the exception of the "single" and "none" index, all of these files serve as an intermediary between the chunk code and the associated index package, providing a dataset chunk index by defining a `H5D_chunk_ops_t` struct, while also defining one or two clients for the index using its associated client struct. All indexes except v1 b-trees use a separate index client for filtered and unfiltered chunks. The "single" index is used when the current and maximum dimensions are equal to the chunk size, so there is always only one chunk. In this case, the index address is simply equal to the chunk address. The "none" index is used for datasets with early allocation, a fixed maximum size, and no data filters. This works similarly to the single index, except multiple chunks are stored at the index location, one after the other in row-major order.

  These chunk indexes thereby allow the chunk code to insert chunks into an index and retrieve these chunks (i.e. their file addresses) given the chunk's coordinates in the dataset. These coordinates are generally "scaled" so that the coordinates are in terms of chunks, not elements. In the scaled coordinate system each chunk has a size of 1 in every dimension.

  The chunk cache is an in-memory cache of the raw data chunks. It uses a very basic design, with a simple hash table with no chaining and a modulus hashing algorithm. Hash value collisions simply

result in the older chunk being evicted from the cache, and otherwise, chunks are evicted as needed to make space from the tail of the LRU (least recently used) list, with a configurable amount of preference given to chunks that have been fully written or read. While it is called least recently used, this is currently a misnomer since, when accessing a chunk that is already in the cache, this chunk is only moved forward one slot in the LRU list instead of being moved to the head. When performing I/O on a cached chunk, the library switches to using the low-level I/O routines for compact datasets, since these routines are built to perform I/O to/from memory buffers. When skipping the chunk cache, the library switches to using the low-level I/O routines for contiguous datasets (skipping the sieve buffer that's present in the higher-level routines), performing I/O to each chunk directly on disk from the application's memory buffer. The library will skip the chunk cache if the total size of the cache is smaller than a single chunk or if accessing the file in parallel with write permissions, but if the dataset has filters or the fill value needs to be written to a chunk, the library will still use the cache temporarily (flushing and evicting the chunk immediately) because only the cache routines have currently been written to handle those cases.

- Virtual datasets

  Virtual datasets allow an application to define a dataset whose actual data is stored in separate "source" datasets. Virtual datasets do not store any data themselves, they consist only of a set of mappings from a selection in the virtual dataset to a selection in a source dataset. These mappings can be standard selections, "unlimited" selections where one dimension has a count or block of `H5S_UNLIMITED`, or "printf" selections, where the file or path name of the source dataset contains a printf-style character sequence that is replaced by a block number, corresponding to the blocks in the selection in the virtual dataset.

  I/O for virtual datasets is performed directly between each source dataset and the application's memory buffer, with no intermediate buffer, using the standard facilities for each source dataset. The virtual dataset code only handles the selection transformations necessary to set up these operations, with the exception that the virtual dataset code also writes fill values to the read buffer when the application reads from a section of the virtual dataset that has no associated source dataset mapping. To perform this transformation, the virtual dataset code uses the `H5S_select_project_intersection()` operation covered in the Dataspaces section. For each mapping, there are four selections involved: the memory selection, the file selection (in the virtual dataset), the mapping's virtual selection, and the mapping's source selection. To understand how to perform the transformation with no intermediate buffer, first consider how things would work if there were an intermediate buffer representing the entire virtual dataset. We'll use a read operation for this description, but it works similarly for write. The data from the source selection would be read from the source dataset to the virtual selection in the virtual dataset. Then the file selection in the virtual dataset is read to the memory selection in the application's read buffer. However, we only care about the elements that are selected in the virtual dataset in both the file selection and the mapping's virtual selection, all other elements will not make it all the way from the source dataset to the memory buffer. We therefore want to map the intersection of the file selection and the virtual selection onto both the memory selection and the source selection. To get the mapping onto the memory selection, we use `H5S_select_project_intersection()` to project the intersection with the virtual selection (source intersect space) within the file selection (source space) onto the memory selection (destination space). To get the mapping onto the source selection, we use `H5S_select_project_intersection()` to project the intersection with the file selection (source intersect space) within the virtual selection (source space) onto the source selection (destination

space). We then use the resulting two selections in the underlying dataset I/O between the memory space and the source dataset, and repeat this process for every other mapping (that has any intersecting elements).

The unlimited and printf style mappings allow the size of the virtual dataset to depend on the sizes of the source datasets. When the application calls `H5Dget_space()`, the virtual dataset code, in `H5D__virtual_set_extent_unlim()` checks the dimensions (extents) of all the source datasets that are associated with unlimited or printf mappings, then computes the resulting extent of the virtual dataset based on the setting from `H5Pset_virtual_view()`. The library must also generate non-unlimited versions of all the unlimited virtual and source selections that match this size (or the source dataset size in case it is smaller), these are called "clipped" selections. Currently, the library also adjusts the extents of these clipped selections to match the virtual and source datasets, however, this is probably not necessary since the extent is no longer used by `H5S_select_project_intersection()` and this code could be simplified somewhat. This clipping of the selections and adjustment of the extents is also performed in `H5D__virtual_init()`, which is called once to initialize the dataset in memory, though this function does not check the source dataset extents or adjust the virtual dataset extent.

- I/O pipeline

  After the VOL callbacks in `H5VLnative_dataset.c` set up the array of dataset info structs (one for each dataset involved in I/O), code flows into `H5D__read()` or `H5D__write()`, which are the central functions of the I/O pipeline. These functions first initialize the I/O operation, and this initialization has several steps, consisting both of general functions in `H5Dio.c` and the layout `io_init` and `mdio_init` callbacks. The actual I/O follows two paths currently: multidataset I/O and single dataset I/O. The multi-dataset path is followed only in parallel when not using selection I/O, and passes into the MPIO-specific functions in `H5Dmpio.c` via the `multi_read/write_md` callback. This branch is only taken when not using selection I/O, and will likely be skipped in the future for all cases except filtered parallel I/O. See the parallel and selection I/O sections for more information. The other branch loops over every dataset, making the `multi_read/write` callback for each one.

  One potential point of confusion in the I/O pipeline is that there are two closely related I/O callback layers: `io_ops` and `layout_ops`. The `multi_read` and `multi_write` callbacks in `io_ops` are set to point to the `ser_read` and `ser_write` callbacks from the dataset's `layout_ops`, while `single_read` and `single_write` are set to either the direct I/O operations in `H5Dselect.c` or the type conversion operations in `H5Dscatgath.c`. There are also `md_io_ops` callbacks which are used instead when performing collective I/O without selection I/O. The `io_ops` and `md_io_ops` layers have probably outlived their usefulness and could probably be eliminated.

  `H5D__read` and `H5D__write` call `multi_read`/`multi_write`, which again point to `ser_read`/`ser_write`. The layout then performs any layout-specific operations necessary (chunked will assign other layout callbacks to handle the actual I/O, virtual will make whole new internal I/O calls, compact simply uses the contiguous callback), then calls the appropriate `single_read`/`single_write` callback, which unpacks the selection, handles any type conversion, and calls the layout's `readvv`/`writevv`, which performs the actual I/O using vectors of offsets and lengths in memory and the file.

- Parallel and Collective I/O

When not using selection I/O, parallel collective I/O is implemented using the functions in `H5Dmpio.c`. The decision on whether to use these functions is made in `H5D__mpio_opt_possible()`, as called by `H5D__ioinfo_adjust()`. First, at the end of `H5D__read()`/`H5D__write()`, the library loops over the datasets, making the `mdio_init` layout callback for each. This callback adds each selected piece to a global skiplist of selected pieces, looking up its address on disk if necessary. In this context, "piece" refers to a chunk or a contiguous dataset. The code in `H5Dio.c` then flows into `H5D__piece_io()` in `H5Dmpio.c` via an abstraction layer and intermediate functions that can probably be removed in the future.

For unfiltered datasets, there are two main raw data I/O algorithms: link piece and multi chunk. The link piece algorithm builds an MPI datatype that spans the entire I/O operation, across all datasets, and performs a single collective MPI I/O call. Link piece I/O is primarily implemented in `H5D__link_piece_collective_io` and makes heavy use of functions in `H5Smpio.c`. The multi-chunk algorithm instead iterates over all chunks in a dataset, performing either collective or independent I/O on each chunk that is involved in I/O sequentially. The multi-chunk algorithm operates on one dataset at a time, and ranks coordinate to exchange information about which chunks are selected on each rank before chunk iteration begins. Contiguous datasets are handled using simple collective I/O when using the multi chunk algorithm. The choice on which algorithm to use can be made automatically or specified using DXPL properties. This decision is implemented in `H5D__piece_io()`. Likewise, the threshold for switching between collective and independent for each piece can be specified using a DXPL property. For unfiltered I/O the resulting MPI datatypes are passed to the MPIO VFD by placing them inside the DXPL under a special, undocumented property. This means that unfiltered parallel collective I/O only works with the MPIO driver when not using selection I/O, and outside developers cannot easily take advantage of HDF5's full parallel capabilities with any other file driver in this case. We plan to eliminate this pathway as soon as we are certain that all current capabilities are covered by selection I/O and no case will see a significant performance degradation.

Filtered I/O is always handled in H5Dmpio.c, and requires collective access for writing (though the low-level I/O can be switched to independent using `H5Pset_dxpl_mpio_collective_opt`). Filtered I/O also uses both link chunk (piece) and multi-chunk algorithms. For read operations, these algorithms are fairly simple and similar to their unfiltered equivalents. Link chunk uses vector I/O instead of selection I/O to read the filtered chunks from disk. For multi-chunk, each rank iterates only over chunks it has selected, and takes part in a collective read for each, or participates in a collective read with 0 bytes if it has run out of chunks but another process still has chunks. For write operations, filtered parallel I/O is more complicated. The algorithm is covered in more detail in the function header comments for `H5D__multi_chunk_filtered_collective_io()` and `H5D__link_chunk_filtered_collective_io()`, but briefly summarized: first each chunk is assigned an owner process, then chunks that are being partially written to are read from the file, then processes send their writes to each chunk to that chunk's owner process, then the owner applies updates from all processes and filters the chunk, then the chunks are all collectively (re)allocated, written, and collectively inserted in the chunk. The main difference between multi chunk and link chunk in this case is that multi chunk performs the actual write on a chunk-by-chunk basis similarly to the read case, while link chunk issues a single vector write call.

Independent I/O is handled in the same way as serial I/O, without entering `H5Dmpio.c`. It is important to keep in mind that I/O can switch from collective to independent at multiple places in the I/O pipeline. The function may have been called independently, in which case no collective coordination may occur,

or it could have switched to independent early due to a case being unsupported in collective, or it may use the collective I/O pathways in the library but use independent MPI I/O calls due to a property list setting requesting the library do so.

- Selection I/O

  Selection and vector I/O are facilities that allow the library to pass information on non-contiguous I/O to the file driver layer, either a list of HDF5 dataspace selections or a simple list of offsets and lengths. This eliminates the need to pass an MPI datatype to the file driver, and allows file drivers to more intelligently handle non-contiguous I/O. More work is needed to allow external file drivers to fully support collective I/O however.

  In the I/O pipeline, the decision on whether to use the selection I/O pipeline is made during the various initialization steps. Any dataset's layout `init` callback can disable selection I/O, as well as several global triggers within the initialization code in `H5Dio.c`. The main selection I/O pipeline follows a similar path to the serial/independent pipeline, with some changes to support selection I/O. Though it follows the independent code path, selection I/O still supports full MPIO collective I/O, since the datatypes are constructed in the MPIO file driver using the selections. When operating on a single dataset, the primary change is that the low-level I/O call changes from a series of scalar (single offset/length) calls to a single selection I/O call, built up from all the selections within all the selected chunks in the dataset, and the accompanying memory selections. When operating on multiple datasets, the layout's `read`/`write` call does not actually perform I/O, instead it adds its selections, offsets, and buffers to a global list. At the end of `H5D__read()`/`H5D__write()`, the library then calls `H5F_shared_select_read()`/`H5F_shared_select_write()` with these lists.

  The layout I/O callbacks also avoid performing the actual I/O in the case of type conversion, even if there is only one dataset involved. When using type conversion with selection I/O, at the end of `H5D__read()`/`H5D__write()`, the library calls `H5D__scatgath_read_select()`/`H5D__scatgath_write_select()`. These functions then implement type conversion while performing I/O to/from disk using the selection I/O interface. These functions require that either the type conversion buffer is large enough to fit the entire I/O operation, or that in-place type conversion is used (or a combination of the two), since they are intended to maximize the use of selection I/O and minimize the number of VFD I/O calls. This behavior also makes it possible to support collective I/O with type conversion, a feature that is only supported with selection I/O.

- Allocation

  Space for the dataset's raw data can be allocated when the dataset is created (`H5D_ALLOC_TIME_EARLY`), when it is first written to (`H5D_ALLOC_TIME_LATE`), or, in the case of a chunked dataset, allocate each chunk when only when that chunk is written to (`H5D_ALLOC_TIME_INCR`). For unfiltered datasets where the file is open with write access in parallel, early allocation is forced on.

  The code path for early allocation is somewhat convoluted and is worth covering here. The central function for dataset allocation is `H5D__alloc_storage()`, which is called on dataset creation with early allocation via `H5D__update_oh_info()` and `H5D__layout_oh_create()`. This function works in two phases and uses switch statements to handle different layouts differently, instead of implementing a layout callback interface like other operations. For contiguous datasets, the first phase allocates space in the file while the second phase writes the fill value if appropriate. For compact datasets, the first phase allocates a memory buffer for the data and the second phase writes

the fill value. For chunked datasets, the first phase allocates the root of the chunk index using `H5D__chunk_create()`, and the second phase allocates all the chunks (and adds them to the index) using `H5D__chunk_allocate()` (called by `H5D__init_storage()`) if the allocation time is not `H5D_ALLOC_TIME_INCR`. Virtual datasets do nothing here since they store no data of their own.

The "none" and "single" indexes work differently for allocation since they don't have a real index, and there is code in `H5Dchunk.c` added specifically to support the none index, subverting the index callback layer. The none index, since it is only used with early allocation, allocates space for all chunks in the index `create` callback, as called by `H5D__chunk_create()`. Then later, when allocating chunks in `H5D__chunk_file_alloc()`, as called by `H5D__chunk_allocate()`, the library checks for use of the none index and, if it is in use, avoids actually allocating space for the chunk since it was already allocated in the `create` callback. The single index inverts this, and does not do anything in the `create` callback, leaving the index address undefined, and instead fills in the index address (with the address of the chunk) when the (single) chunk is allocated.

4.1.13. Datatypes (`H5T`)

- In-memory objects

  Opened handles to datatypes are represented by the top-level `H5T_t` struct, which points to the underlying `H5T_shared_t` struct. The shared struct stores information associated with the datatype itself, while the top-level struct stores information about the location the datatype was accessed through.

  If an in-memory datatype object is a handle to a committed datatype, that committed datatype object will be accessible through the `vol_obj` pointer on the top-level datatype struct. For the native VOL, this VOL object's 'data' field will be a copy of the datatype's `H5T_t` structure.

  Committed datatype objects can be removed from storage with `H5Ldelete()` (formerly `H5Gunlink()`). Although the datatype cannot be directly opened after this call is made, the library will not actually free the datatype's resources while other objects that reference it still exist.

- Modification of Datatypes

  Control of datatypes' modifiability is implemented through the state field `H5T_state_t` on the shared datatype structure. There are five datatype states, each of which is summarized in Table 4.3

Datatype State	Description
Transient	Transient datatype. Modifiable and closable by user.
Read-only	Transient datatype. Not modifiable, but may be closed by user.
Immutable	Transient datatype. Not modifiable or closable by user. Predefined types are created with this state.
Named	Committed datatype that is not open in-memory.
Open	Committed datatype that is open in-memory.

Table 4.3. – Datatype states

Using `H5Tcopy()` on a datatype always returns a copy with a transient, modifiable state.

- Datatypes of objects

  If an object is created with a transient datatype, the objects stores a datatype message describing the datatype in its object header. If an object is created with a committed datatype, then the object stores a pointer to the committed datatype's object header. If multiple objects share the same committed datatype, the same object in storage is shared between them.

  For objects with transient datatypes, functions that get the datatype associated with an object, such as `H5*get_type()`, return a copy. For objects with a committed datatype, these functions return a transient reference to their committed datatype.

- Datatype derivation: atomic vs. composite

  Atomic strings and opaque types are created through `H5Tcreate()`. Other atomic datatypes are derived by using `H5Tcopy()` on predefined types. The derivation of a composite datatype varies by which category of composite is being created.

  Compound datatypes are created through `H5Tcreate()`, followed by the insertion of individual fields with `H5Tinsert()`. The compound datatype struct, `H5T_compnd_t`, stores information about its member datatypes, `H5T_cmemb_t`, in a contiguous array. New members are appended to the end of the array.

  Other composite datatypes are created, with a specified 'base' type, through the dedicated API functions `H5T[array/vlen/enum]_create()`. The base type is stored in the `parent` field of the shared type information structure. If a datatype is itself a base type, then its `parent` field is `NULL`.

  The atomic or composite nature of a datatype is stored in its shared `type` field, which specifies which field in the shared type information structure's union `u` is meaningfully populated.

- Native datatypes & standard datatypes

  The library's predefined 'native' datatypes match the size and endianness of the platform that the library is built on. Thus, on different machines or operating systems, the same native datatype may correspond to a different series of bytes in storage.

  The library's 'standard' datatypes each have a defined endianness and size that is included in their name. For example, the type `H5T_STD_U16BE` will always be a 16-bit in size and have a big-endian byte order, regardless of system architecture.

  Native predefined datatypes are first defined in `H5T__init_native_[float_types/internal]()`, and non-native predefined types are derived from them afterwards during initialization of the datatype module.

  The macros used to access predefined datatypes evaluate to global library IDs, with a check that ensures the library is initialized before any operations are performed.

- Datatype life cycle (CRUD)

  The API-exposed aspects of the datatype life cycle of are summarized in Figure 4.3.

  Transient datatypes are created by a function of the form `H5T*create()` or `H5Tcopy()`. Copies of transient datatypes are returned from copying datatypes or querying the datatypes of objects with transient datatypes.

While the API's `H5Tcopy()` always create a datatype with the transient state, the internal `H5T_copy()` may, depending on the parameters it is given, perform a full copy resulting in a non-transient datatype.

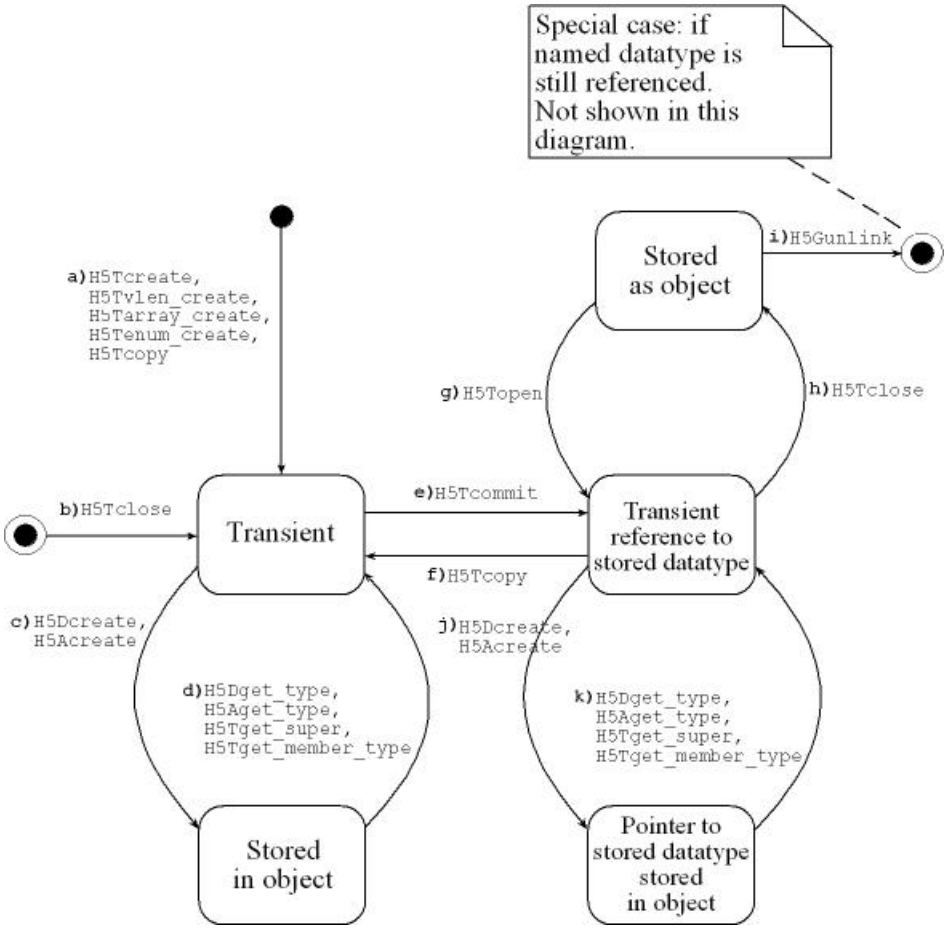


Figure 4.3. – Datatype life cycle

If a transient datatype is committed, the `vol_obj` field of its shared information struct is populated with a copy of itself that is stored on disk, and the datatype's handle becomes a transient reference to the stored datatype. A transient reference to a stored datatype is also obtained when querying the datatype of an object with a committed datatype.

A committed datatype will be slated for removal from the file by `H5Ldelete()` (or the deprecated `H5Gunlink()`) if no other objects reference it. If it is unlinked and the objects that reference it are later deleted, the datatype's memory will be marked as free by the free space manager at that time.

- Alignment

  Different platforms may have different byte alignment restrictions when placing types within structures. The `HOFFSET` macro should be used when creating a compound type in order to align the inserted fields with any automatically added bytes of padding.

Alignment of stored fields is not necessary when storing data structures on disk, and so the padding bytes can be removed to increase storage efficiency by 'packing' the datatype. Compound types can be packed with `H5Tpack()` to create storage-optimized unaligned versions. `H5Tpack()` directly modifies the datatype it is given, so it should usually be preceded by `H5Tcopy()`. Note that a packed datatype will usually be unable to describe in-memory data.

- Endianess

  A platform that stores the most significant byte of a number at the smallest memory address is 'big-endian', and a platform that stores the least significant byte at the smallest memory address is 'little-endian.' The HDF5 library determines during its configuration which convention the current platform uses, and stores that as the 'native' convention, used by predefined types such as `H5T_NATIVE_INT`.

- Datatype hard and soft conversions

  Each datatype conversion function has a 'hard' or 'soft' persistence. The persistence of a conversion function determines what source and destination datatypes it will be used for. A hard conversion function will only be used for the specific datatypes it is registered for, while a soft conversion function will be applied to any datatypes that match the classes of the source and destination types it is registered for.

  A new hard conversion function will overwrite any previous conversion functions for the provided types, and a new soft conversion function will overwrite the conversion functions for any types to which it is applicable that do not already have a hard conversion function.

- Variable length datatypes

  Because the total amount of storage space needed for objects with a variable length datatype cannot be known ahead of time and may change, data with a variable length type is stored in the file's global heap, rather than in the data object. The metadata describing the variable-length type itself is stored in a Datatype Message in an object's header, just as for non-variable-length datatypes.

  As a derived type, a variable-length datatype stores a pointer to its base datatype in the `parent` field of its shared type information structure.

- Reference datatypes

  Reference types are internally handled slightly differently than other 'complex' datatypes (e.g. compound, array, variable length types). Rather than having a unique structure which is a possible value of the union `u` in the shared type information structure, reference datatype objects are considered 'atomic', and are represented by a `H5T_atomic_t` instance. Reference-specific information is then stored under the `r` field of the atomic datatype structure's own union `u`.

- Datatype serialization

  Datatype serialization to and from datatype messages is performed by the encode/decode callbacks on the datatype message class, `H5O__dtype_[encode/decode]()`. It is also exposed as part of the API through `H5Tencode()/H5Tdecode()`. For atomic datatypes, encoding and decoding are accomplished non-recursively. For compound types, a separate recursive call encodes each member field of the overall datatype. For arrays, variable-length types and enums a single recursive call is used to encode the base type.

- Addition of new native types

While the library allows applications to create their own custom datatypes, there are some reasons why a native datatype may be desired. For example, two applications may make different choices for how to implement complex numbers as a compound datatype, decreasing the portability of the data. For those interested in the process of introducing new native datatypes to the library, RFC THG 2015-04-29 [19], specifically section 3.1, details which parts of the library and tools must be changed for compatibility with a new native datatype.

4.1.14. Data Filters (H5Z)

- Data filter pipeline

  Data filters are used to manipulate data. The HDF5 library provides several built-in filters to allow for compression (H5Z_FILTER_DEFLATE, H5Z_FILTER_SZIP, H5Z_FILTER_NBIT, H5Z_FILTER_SCALEOFFSET), shuffling (H5Z_FILTER_SHUFFLE), and error checking (H5Z_FILTER_FLETCHER32). While filters can be used individually, they can also be stacked into a filter pipeline. Filters for contiguous datasets are not currently supported because implementing random access for partial I/O is difficult, and compact datasets are also not supported because the potential gains from filters are negligible on such small datasets. Theoretically, the maximum number of filter applications should be unlimited, but because each filter uses a bit in a 32-bit field, the actual maximum number of filter applications is 32.

  Users can also define their own filters to customize data processing further, which provides a high degree of flexibility for tailoring the pipeline to specific needs. As long as the filter is associated with a dataset upon the creation of that dataset, can be used with chunked data (H5D_CHUNKED), and is applied to each individual chunk in a H5D_CHUNKED dataset, then the custom filter may be registered and used. This is done through a two-step process. First, three callback functions must be written. The first is H5Z_can_apply_func_t, which checks against the dataset's dataset creation property list, datatype, and dataspace. The next is H5Z_set_local_func_t, which sets local flags and fields using the aforementioned three parameters. The last is H5Z_func_t, which takes in a bit vector of flags, the number of elements in the array of auxiliary data for the filter, the array of auxiliary data itself, the number of bytes in the filter buffer, the size of the filter buffer, and the filter buffer itself, and uses the fields set by the previous method to return the altered buffer back to the user along with its size.

  Once the above callbacks have been defined, the filter must be registered using H5Zregister(). H5Zregister() takes a pointer to a buffer for the class data structure and returns a H5Z_class_t (a two-byte identification number), which can be either H5Z_class1_t or H5Z_class2_t.

  H5Z_class_t is a macro that maps to either of the above two versions depending on the needs of the application. H5Z_class_ts start from 0-255, and these values are reserved for official HDF5 filters such as the ones listed above. 256-511 include the numbers that may be used for temporary testing. Lastly, 512-65535 is reserved for future assignments. If the filter is set as optional, then the library will skip the filter while proceeding through the filter pipeline.

  At the center of the data filter pipeline is H5Z_pipeline(), which is what carries out the pipeline. This method processes filters in definition order, or reverse order if the H5Z_FLAG_REVERSE flag is set. It takes in a H5O_pline_t, an unsigned int for flags, an array of unsigned ints for a filter mask, a H5Z_EDC_t, an H5Z_cb_t instance, a size_t nbytes, which holds the number of bytes to filter, a size_t buf_size, which holds the total allocated size of the buffer, and buf, which is the buffer itself. This method will first check if the filter is registered, and if it is not already registered, it will try to load

it dynamically and then register it. Once this is done, it will go through the list of filters in definition order and attempt to apply each filter to the data.

It is important to note that functions that allocate and free memory must be treated with care, since if there is a mismatch between the memory allocators used in the library and any memory that might be moved around by a filter, there may be problems. This issue can be resolved via the `H5allocate_memory()` and `H5free_memory()` functions, but if these methods are used, then the filter would have to be linked to a specific library version, which may be problematic.

- Data transforms

  Data transforms are also included in `H5Z`, and can be accessed via methods like `H5Z_xform_eval()` on trivial transformations or `H5Z__xform_eval_full()` on complex transformations. This is done through the use of `H5Z_node` to hold the parse tree and `H5Z_token` to hold the original expression, current token values, and previous token values. Each `H5Z_node` contains a left child, right child, `H5Z_token_type` to hold the type, and a `H5Z_num_val` to hold the value. Additionally, it includes macros for several different operations such as `H5Z_XFORM_DO_OP1`, each of which is used in `H5Z__xform_reduce_tree()` to simplify parse trees, which is used when evaluating transformations.

4.1.15. Dataspaces (`H5S`)

A dataspace in HDF5 consists of two parts: an extent and a selection within that extent. They are mostly handled separately in the HDF5 library and only tied together at the top level of the structure. Extents are fairly trivial, consisting of a rank (number of dimensions) and size in each of these dimensions.

Selections are handled inside the `H5S` (dataspace) package with a flexible, class-like interface consisting of a `struct` of function pointers which must be implemented by each selection type. Currently, four selection types are implemented in HDF5: none, all, hyperslab, and point. 'None' refers to a selection with no elements selected, while 'all' refers to a selection of every element in the extent. Hyperslab selections are created using combinations of (`N`-dimensional) rectangular blocks, and a point selection is a list of individual elements. Point selection is the only selection type where the elements may be selected out of order, that is, the order of elements within the selection might not match the order of those elements within the serialized extent. Hyperslab blocks may be added in any order, but after combination, the order of the elements within the selection matches the order within the extent, as shown in Figure 4.4. The order of elements within the extent is the same as the ordering of memory elements in a multi-dimensional array for the programming language used. Examples in this section use row-major ordering, as used by C.

- Dataspace in-memory objects

  The dataspace object in HDF5 is represented with the struct `H5S_t` containing two fields, `extent` and `selection`, exactly mirroring the conceptual model above.

  The extent field contains the dataspace extent (number of dimensions and number of elements in each dimension), as well as the type of extent, number of elements (for convenience), and information needed to encode the extent to an object header message.

  The selection field contains the selection class (type) struct, which consists of a list of function pointers that the selection class uses to implement the necessary selection callbacks. The selection field also

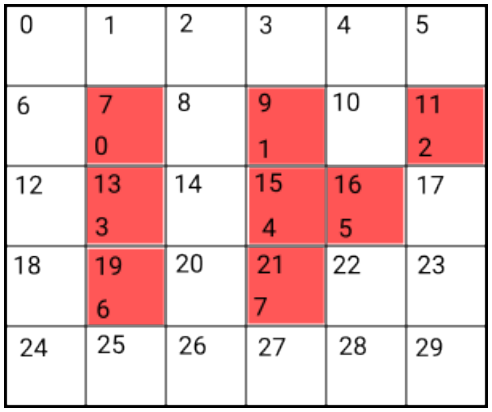


Figure 4.4. – A dataspace extent and selection (red) within the extent. The serialized offset within the extent is in the upper left of each element and the serialized offset within the selection is in the lower left.

contains information on the selection offset, number of elements selected (again for convenience), and the information describing the actual shape of the selection, which depends on the selection class.

- Dataspace extents

  Dataspace extents can be one of three types: scalar (`H5S_SCALAR`), simple (`H5S_SIMPLE`), and null (`H5S_NULL`). A scalar dataspace extent represents a single element, a simple dataspace extent represents an N-dimensional array of elements, where N is between 1 and 32, and a null dataspace represents no elements. Simple dataspaces have an array size in each dimension, and also a maximum array size in each dimension (or `H5S_UNLIMITED` to represent no maximum size). The size of each dimension can be changed with a call to `H5Sset_extent()`, though each dimension size cannot be set to larger than the corresponding maximum dimension size. The maximum dimension sizes and the rank (number of dimensions) cannot be changed.

- Dataspace selections

  Dataspace selections are used to select a subset of elements within the dataspace extent. They can be used to perform an operation on these elements, or simply to describe the subset (such as with a region reference). There are multiple types of selections, each of which must be defined by implementing the selection class, consisting of a series of callback functions that must be implemented. In addition, there are other functions that deal with individual selection types directly (such as `H5Sselect_hyperslab()`) or that deal with multiple selection types (such as `H5Sselect_project_intersection()`).

- Hyperslab selections

  Hyperslab selections represent an N-dimensional hyperrectangle of elements within the extent (where N is equal to the rank of the extent), a regular pattern of these hyperrectangles, or a combination of multiple hyperrectangles and/or patterns.

  Hyperslabs are primarily constructed through the API using `H5Select_hyperslab()`, which takes, as parameters, `start`, `stride`, `count`, and `block`, all of which are arrays of length matching the rank of the dataspace extent. `start` indicates the element at which the hyperslab begins (i.e. the element closest to position 0 in each dimension). `stride` indicates the spacing between the starting

position of regularly spaced blocks in each dimension. `count` indicates the number of regularly spaced blocks in each dimension. `block` indicates the size of each regularly spaced block. In addition, hyperslabs can be combined in different ways using the `op` parameter, potentially yielding hyperslabs that cannot be represented using a single set of `start`, `stride`, `count`, and `block` arrays.

Internally, hyperslabs are represented in one of two ways. Hyperslabs that can be represented using a single set of `start`, `stride`, `count`, and `block` arrays are called *regular* hyperslabs and are represented using these arrays. Other, irregular, hyperslabs are represented using a data structure called a *span tree*.

A span tree is a multidimensional linked-list representing spans of selected elements. It is implemented using the `H5S_hyper_span_info_t` and `H5S_hyper_span_t` structs. The entry point for a span tree is a `H5S_hyper_span_info_t` which is the head of the linked list of spans (each represented by a `H5S_hyper_span_t`) in the slowest changing dimension of the dataspace. Each span contains a `down` pointer to a `H5S_hyper_span_info_t` in the next fastest changing dimension.

To more easily understand span trees, it is useful to start at the fastest changing dimension. In this dimension, there are no `down` pointers, and each span consists only of a low and high bound (coordinates in the fastest changing dimension of the dataspace), and a pointer to the next span. This forms a one-dimensional pattern of "dashes". In the next dimension up, this process is repeated, except instead of each span selecting a list of elements, it selects a list of these dashed patterns. An example of a span tree representation of a selection in a 2-D dataset can be found in Figure 4.5.

Hyperslabs can have either a regular description (start, stride, count, and block), a span tree description, or both. Operations that are performed on hyperslabs need to be able to handle all of these cases. Some operations will construct a span tree for a regular on entry in order to simplify implementation (so it only needs to be implemented for span trees), though if an operation can be performed on a regular hyperslab directly that is generally more efficient. If a span tree is changed for a regular hyperslab it must be assumed it is no longer regular so the start, stride, block, and count are marked invalid. Also, after operations that can change a hyperslab, they will generally check if the hyperslab is regular and compute valid start, stride, block, and count values if so.

- Point selections

  Point selections are a simple linear list of individual elements that are selected within the dataspace extent. They are implemented using a simple linked list of N-dimensional coordinates of each point. While conceptually simple, the fact that the points can be selected out of order with respect to the ordering of elements within the extent causes complications. The parallel code needs to take special care to reorder point selections as needed to obey MPI semantics, and virtual datasets do not support point selections at all. It is worth noting that it would not take much more work to implement point selection support for virtual datasets.

- Selection iterators and iteration operations

  A selection iterator is a structure used to perform operations that iterate over a dataspace, such as generating offset/length pairs of selected elements (`H5S_get_seq_list()`) or making a callback for each element selected (`H5S_select_iterate()`. The selection iterator includes another generic interface, `H5S_sel_iter_class_t`, that contains callback functions that must be implemented by each selection type. In addition, hyperslab and point selections store their own data in the selection iterator struct. These callbacks are used by the higher level code in `H5Sselect.c` to implement these operations. The iterator operations can also be called directly by external packages.

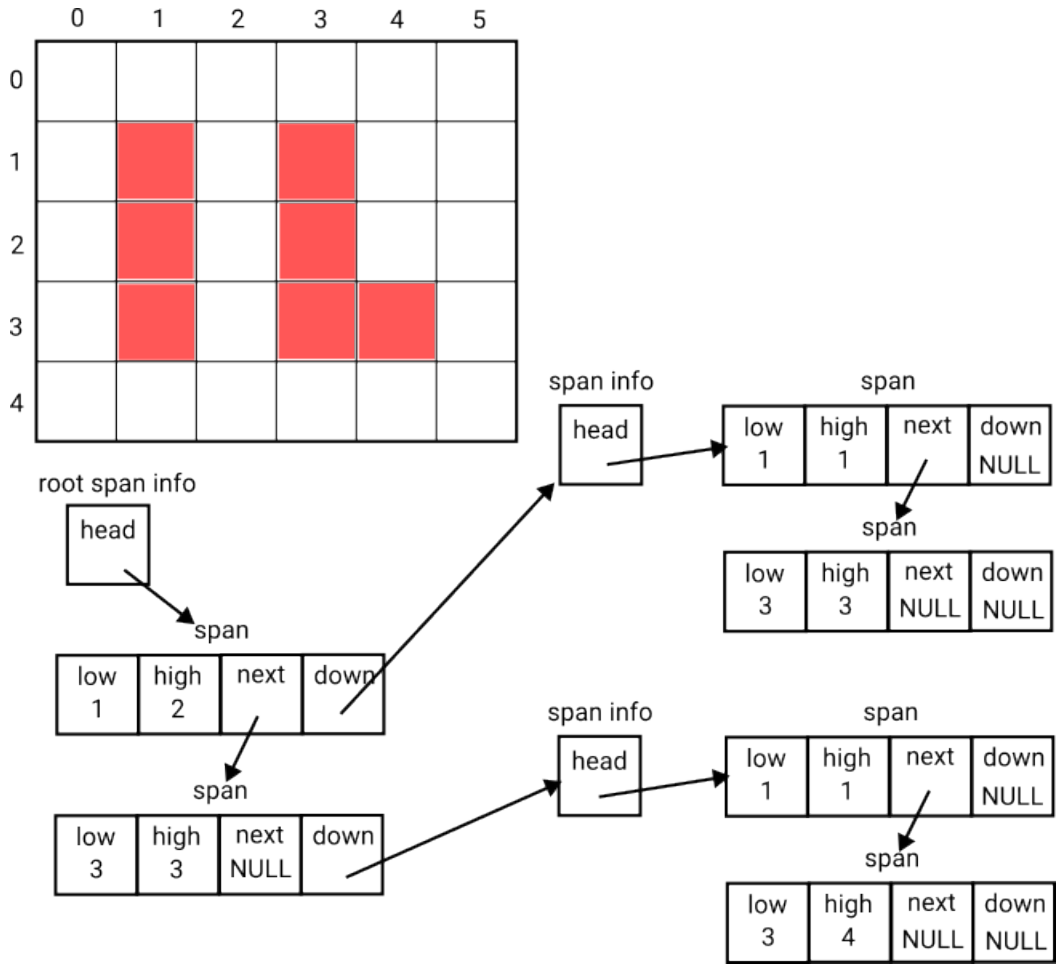


Figure 4.5. – An illustration of span trees for a 2-D selection.

- Projections and the "shapesame" case

  When performing I/O with selections, the library is given an extent and selection for the memory buffer, and a selection for the file (the extent for the file is implied by the dataset size). For contiguous datasets, the generic case here can be easily handled using selection iteration. The elements selected in the memory can be matched up 1-to-1 with elements selected in the file by the order within the selection, so the Nth element selected in memory is written to/read from the Nth element selected in the file. For chunked datasets, however, this becomes more complicated. The dataset in the file is broken up into chunks that may not be contiguous within the serialized (flattened) extent of the dataset, but the memory buffer is not broken up in the same way (indeed, the memory buffer need not have the same extent or even rank as the dataset). We therefore need to break the file selection up into a series of selections, one for each chunk. It is easy to do this using selection interfaces that compute the intersection of two selections. However, we also need to compute the selection in memory that corresponds to the selection within each chunk. Since the chunks may not be contiguous in the serialized extent, we can no longer rely on the 1-to-1 mapping used for contiguous datasets.

However, there is a shortcut we can take when a certain common condition is fulfilled. Often, the selections in the memory and file have the same shape, possibly with some offset between the two. If this is the case, we can simply, for each chunk, compute the intersection ("and") of that chunk with the file selection, then copy that intersection to the memory space, and apply any offset computed from comparing the overall file and memory selections. This copied intersection, called a projection, is now a selection containing the memory elements that match, in the correct order, the file elements selected that are in the current chunk. The dataset code then uses these file and memory chunk selection pairs to construct the I/O operation. The function used to check if two selections have the same shape is `H5S_select_shape_same()` (wrapped by the macro `H5S_SELECT_SHAPE_SAME()`), and the function used to construct the projected dataspace is `H5S_select_construct_projection()`.

While the selections must have the same shape for this case to work, it is actually possible for the extents to be different ranks. If we have selections A and B, with extents of rank M and N, respectively, where $M > N$, then A and B have the same rank if and only if selection A spans only a single element in the slowest changing $M - N$ dimensions, and if A and B are identical in the fastest changing N dimensions, possibly with a constant offset. The library does not currently try to identify cases where the selections could be topologically identical by unrolling one or more dimensions, as this would require a different algorithm to construct the projection. They must actually be the same in the shared dimensions.

- The project intersection operation

  The operation `H5Sselect_project_intersection()` and its corresponding private interface, `H5S_select_project_intersection()`, is robust but can be confusing due to its complexity. This function takes three dataspaces, each with a selection, and outputs a fourth. Internally, this function is currently only used to implement virtual datasets, but it is also used by some VOL connectors to implement generic I/O with selections on chunked datasets. It may be beneficial in the future to implement chunked I/O using this operation when the shapesame case does not apply.

  In the general case for chunked dataset I/O we cannot simply copy the intersection from file to memory as in the shapesame case. The projected chunk selection in memory can be of a totally different shape from the chunk selection in the file. Instead, we can, for each chunk, calculate the offsets of the selected elements in that chunk within the serialized file selection, then select the elements in the memory space with the matching offsets within the serialized memory selection. This resulting memory chunk selection, which is a subset of the overall memory selection, contains the elements selected in the chunk being processed. An example showing this process for a single chunk of a two-dimensional dataset is shown in Figure 4.6. This process is repeated for every chunk that contains any selected elements.

  The general chunked I/O case is supported by `H5S_select_project_intersection()`, though that function also supports other use cases. `H5S_select_project_intersection()` does not require that the selection representing the chunk in the above example be rectangular, it can be any selection that can be represented using the HDF5 API. In this function's terminology, the file selection (in the example) is referred to as the "source space", the memory selection is referred to as the "destination space", and the chunk selection is referred to as the "source intersect space". While the above description implies an element-by-element algorithm for calculating the source intersect space, in the case of hyperslabs (possibly combined with an all selection), the library uses span trees to be able to process the selection multiple elements at a time, which is generally much quicker. The project intersection operation can be thought of as projecting the intersection of the source space and the source

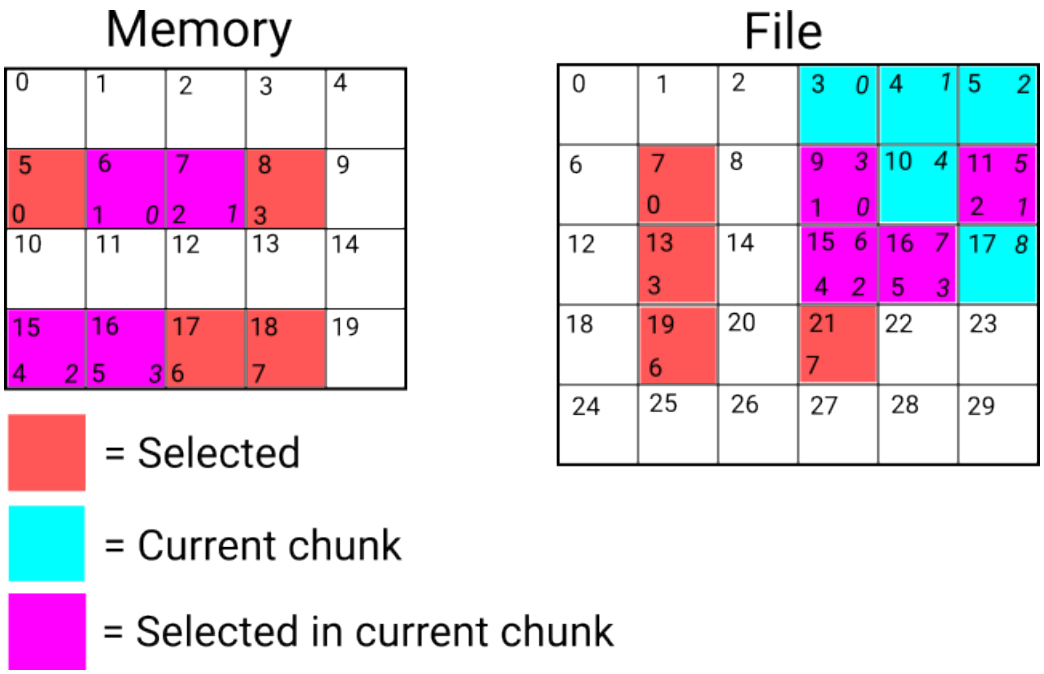


Figure 4.6. – A project intersection operation used for chunk I/O. The serialized offset within the extent is in the upper left of each element, the serialized offset within the selection is in the lower left, the serialized offset within the chunk is in the upper right, and the serialized offset within the intersected space is in the lower right. The result of the operation is the "selection in current chunk" (pink) in memory.

intersect space onto the destination space, where there is a one-to-one mapping between the source space and the destination space (though they need not have the same shape).

This facility is not currently used by the native library to perform chunk I/O. If the shapesame case does not apply, the library iterates element by element over the selections in the dataset package. However, we would like to investigate using this function for the non-shapesame case in the native library, as we believe it may improve performance. The library currently uses this facility to implement virtual datasets, and some VOL connectors use it to implement chunked I/O. At some point in the future, we may wish to unify the shapesame projection and the project intersection operations.

- Parallel (MPI) operations

  The dataspace code is also used to translate dataspace selections into MPI datatypes, which is central to the MPI I/O implementation. This functionality is present in `H5Smpio.c` and is called by functions in `H5Dmpio.c` as well as directly from the MPIO file driver in `H5FDmpio.c`. These functions are mostly straightforward, with different algorithms for point, regular hyperslab, and hyperslab span tree selections.

Source File	Description
H5Spublic.h	Public dataspace API
H5Sprivate.h	Private header (internal dataspace API)
H5Spkg.h	Package header (package private symbols)
H5Smodule.h	Package initialization boilerplate and User's Guide
H5S.c	Public API functions, package initialization, other core functions
H5Sall.c	'All' selection type
H5Sdbg.c	Debugging functions
H5Sdeprec.c	Deprecated functions
H5Shyper.c	'Hyperslab' selection type
H5Smpio.c	Parallel functionality for constructing MPI datatypes
H5Sall.c	'None' selection type
H5Spoint.c	'Point' selection type
H5Sselect.c	Operations on general hyperslab selections. Often just passes down to the specific selection type
H5Stest.c	Functions used only by the regression test suite

Source files in the H5S package

4.1.16. Virtual Object Layer (H5VL)

The virtual object layer (VOL) is an abstraction layer that intercepts all I/O-related calls in HDF5 and forwards them to custom callbacks. VOLs take as input high-level knowledge of the data format and data access characteristics, enabling the development of custom data operators, application-aware caching and logging, monitoring algorithms, specialized file formats, and integration with new storage technologies. Applications depending on HDF5 can leverage VOL plugins to meet their specific data requirements without code changes through the use of environment variables and dynamic linking. This combination of extensibility and transparency positions HDF5 as an enduring data storage solution capable of adapting to a diversity of workloads and future innovations in storage technologies.

VOL connectors are highly flexible in the functionality they can encompass. Connectors can be applied in both serial and parallel HDF5, unlike VFDs. There are two general categories of VOL connector.

1. **Pass-through VOL Connectors**: Forward data to a future VOL connector. An example is context-aware data caching which can intelligently place or prefetch data without change in the overall file format. Alternatively, an adaptive compression makes use of the data format to intelligently select a compression library. Note that HDF5 Filters have some similarities to pass-through connectors, but filters are limited to the native VOL. It is non-trivial to reuse filters in custom connectors. Other examples include the async and external VOLs in HDF5.
2. **Terminal VOL Connectors**: The final VOL connector in a list of VOL connectors. Intended to be used for low-level data storage. An example is the HDF5 native VOL, which constructs the well-known

HDF5 (*.hdf5) file format in storage. Other examples include the DAOS and S3 VOLs. Both DAOS and S3 are object stores, differing from traditional POSIX-compliant filesystems. HDF5 can reliably store and retrieve data in these systems.

VOL connector authors should be aware of the concurrency model of HDF5 when developing their VOLs:

1. **Multi-Process Concurrency**: In parallel HDF5, Not all VOL functions are compatible with MPI_Barrier. This is because not all I/O performed in HDF5 is collective. I/O is collective only when the application specifically passes the `H5FD_MPIO_COLLECTIVE` flag or when the VOL function is related to internal HDF5 metadata updates.
2. **Multi-Thread Concurrency**: VOL connectors will never be called from multiple threads at the same time in HDF5. This is because HDF5 uses a global lock when performing I/O. All VOLs in the list of VOLs will be executed in sequence before the lock is released. To improve concurrency, a VOL connector can leverage asynchronicity and internally spawn and manage its own worker threads for background tasks.

The VOL Class Structure

VOL connectors are defined using the VOL class struct (`H5VL_class_t`), which contains function pointers to various HDF5 operation overrides and other pieces of identifying information. VOLs are intended to follow the singleton design pattern. Typically the VOL class is declared as a static global variable in the VOL source file so that VOL functions can access the variable easily without explicitly passing it in. VOLs can choose to leave most functions unimplemented by setting them to NULL.

The VOL class has a few parameters regarding the identification and versioning of the VOL.

1. **VOL connector ID**: A globally unique integer constant of type `H5VL_class_value_t` that can be used to locate the VOL.
2. **VOL name**: A globally unique semantic string that can be used to locate the VOL. While not technically erroneous, spaces in the VOL name should be avoided since it increases the complexity of VOL initialization through environment variables.
3. **VOL connector version**: An integer indicating the version of this VOL. This is mainly intended to help users ensure they are linking against the correct version of the VOL code.
4. **VOL class struct version**: The version of the class struct being implemented. Different HDF5 versions have different class struct definitions. This allows HDF5 to determine whether the VOL was designed for the particular HDF5 version a user application is compiled against.
5. **VOL capability flags**: Various flags indicating the supported functionality of the VOL. This can include information such as support for provenance, hard links and soft links, and attribute references.

VOLs provide two functions for initializing and finalizing VOL classes.

1. **initialize**: a method called to initialize the VOL connector
2. **terminate**: a method called to free data allocated by VOL connector

There are 9 general classes of functions that can be overridden for the HDF5 data model:

1. `H5VL_info_class_t`: Query the VOL connector's custom information. This can include methods for comparing two VOL connectors and allocating/freeing any custom state held by the VOL connector.

2. `H5VL_wrap_class_t`: Only for pass-through connectors. Used to manage the translation between the HDF5 library's internal representation of objects and the user's representation.
3. `H5VL_attr_class_t`: Override the HDF5 attribute APIs (H5A*), such as `H5Acreate()`.
4. `H5VL_dataset_class_t`: Override the HDF5 dataset APIs (H5D*), such as `H5Dread()`.
5. `H5VL_datatype_class_t`: Override the HDF5 datatype APIs (H5T*), such as `H5Tcreate()`.
6. `H5VL_file_class_t`: Override the HDF5 file APIs (H5F*), such `H5Fcreate()`.
7. `H5VL_group_class_t`: Override the HDF5 group APIs (H5G*), such as `H5Gcreate()`.
8. `H5VL_link_class_t`: Override the HDF5 link APIs (H5L*), such as `H5Lexists()`.
9. `H5VL_object_class_t`: Override the HDF5 object APIs (H5O*), such as `H5Ocopy()`.

There are 4 general classes of function that can be overridden relating to infrastructure/services:

1. `H5VL_introspect_class_t`: Container/connector introspection class callbacks.
2. `H5VL_request_class_t`: Asynchronous request class callbacks.
3. `H5VL_blob_class_t`: Blob class callbacks.
4. `H5VL_token_class_t`: VOL connector object token class callbacks.

VOL Registration

It is mandatory for every VOL to implement and register the `H5VL_class_t` struct. This struct outlines the set of methods the VOL overrides. There are various ways to register the `H5VL_class_t` struct. In the library, there are three API routines for VOL registration:

1. `H5VLregister_connector()`: takes as input the class struct pointer directly and registers it with HDF5
2. `H5VLregister_connector_by_name()`: use the plugins interface (H5PL) to locate the VOL by the VOL name
3. `H5VLregister_connector_by_value()`: use the plugins interface (H5PL) to locate the VOL by the connector ID

During registration, the `initialize` function defined in the VOL class struct will be called if it is non-NULL. This function does not take any parameters – except an empty property list. It is also only called once throughout the entire lifetime of the VOL and should be used to initialize any library or service required to be initialized exactly once. To pass variables to `initialize`, VOL authors should define custom environment variables and document them.

In addition, parameters can be passed to the VOL after registration using `H5VLconnector_str_to_info()`, which takes as input a string that is internally parsed by the VOL's custom `from_str` function (defined in `H5VL_info_class_t`). There is no required structure to this string, although it is best practice to avoid spaces and use ':'or ';' as separators for tokens. More complex parameters should be placed in custom environment variables instead. In `from_str`, the parameter string will be parsed into a VOL-specific info struct, which can be accessed, copied, and queried using methods in `H5VL_info_class_t`. An example of a parameter string to initialize a theoretical VOL named PFS that stores HDF5 data in a file on a PFS could be 'username:/path/to/ssh/key:/path/to/file.txt'.

HDF5 can also transparently locate and register a VOL through the `HDF5_PLUGIN_PATH` and `HDF5_VOL_CONNECTOR` environment variables. `HDF5_PLUGIN_PATH` tells HDF5 the directories to search for VOL libraries and `HDF5_VOL_CONNECTOR` holds a string containing the name and parameterization of a particular VOL to load. H5PL will split this string by spaces to determine the (VOL name, parameter) pair. Only a single VOL and its parameters can be specified using the environment variable. If the parameter string contains spaces, anything after the second space will be deleted. An example of `HDF5_VOL_CONNECTOR` for initializing the example PFS VOL could be 'PFS username:/path/to/ssh/key:/path/to/file.txt', where PFS is the VOL name defined in the class struct. H5PL then automatically loads and registers the VOL, implicitly calling `H5VLregister_connector_by_name()` with the VOL name as input and `H5VLconnector_str_to_info()` with the parameter string as input if it is not empty. `H5Pset_vol()` will then also be called to duplicate the info struct using the `copy` method defined in `H5VL_info_class_t` and store in a property list tracked internally by HDF5. Generally, the info struct will be either passed directly to most VOL functions or queried from a property list.

To be compatible with H5PL, VOLs must implement the `H5PLget_plugin_type()` and `H5PLget_plugin_info()` methods. HDF5 automatically searches for these methods and requires their signatures to be exactly correct. An example code is shown in Listing 4.1.

```
1 static const H5VL_class_t H5VL_custom_vol_g = { /** method overrides */ }
2 H5PL_type_t H5PLget_plugin_type(void) {
3   return H5PL_TYPE_VOL;
4 }
5 const void* H5PLget_plugin_info(void) {
6   return &H5VL_custom_vol_g;
7 }
```

Listing 4.1 – VOL plugin discovery code.

VOL Organization

VOLs can be conceptually organized as directed acyclic graphs (DAGs), although there is no specific DAG data structure in HDF5. Two examples of VOL DAGs and how data flows through them are shown in Figure 4.7. EX1 is a list of VOLs that compress, encrypt, and store the transformed data using a terminal PFS VOL. EX2 is a DAG of VOLs that compresses data and replicates the compressed data to numerous data repositories using terminal VOLs for S3, DAOS, and PFS. To create EX1, the parameter string would have to contain the information needed to register and initialize the compress, encrypt, and PFS VOLs. For example, `HDF5_VOL_CONNECTOR`='compress zlib;encrypt:aes:/home/user/key.aes;PFS:/home/user/ex1.h5'. This string would be first passed to compress, where the relevant information needed to initialize the compress module will be parsed (zlib). Within the compress VOL, the name of the next VOL to create will be extracted (encrypt) and `H5VLregister_connector_by_name()` will be called using the remaining parameter string (aes:/home/user/key.aes;PFS:/home/user/ex1.h5). This will repeat recursively until each VOL is registered and defined. There is no specific functionality in HDF5 to initialize multiple VOLs at the same time – it is the responsibility of a pass-through VOL developer to initialize any future VOLs it may depend on.

After initialization, I/O operations (e.g., dataset reads and writes) can be performed. Write operations are typically executed using preorder traversal (left-to-right for lists). For EX1, compression is first applied to

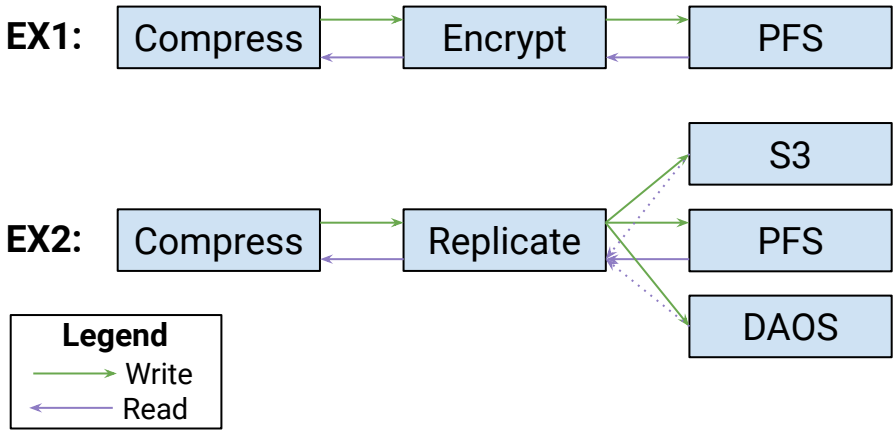


Figure 4.7. – Depiction of data flow in VOL DAGs

the data, then encryption, and then persistent storage. EX2 is similar. Reads are typically executed using postorder traversal (right-to-left for lists). For EX1, data is first read from the PFS, decrypted, and then decompressed. In EX2, the replication VOL decides one of the three destinations (PFS, S3, or DAOS) to read data from, reads the data from that destination and then goes for decompression. In practice, these traversals are executed using recursion, where the first VOL registered is the root of the recursion. In EX1 and EX2, the root is the compress VOL. Pseudocode for the write and read operations for the compress VOL are depicted below in Listing 4.2. For writes, the input buffers stored in `buf` will be compressed into a new set of buffers `compressed`. The maximum size a compressed buffer can be is assumed to be the size of the original data, which is calculated using a custom `get_data_size` function. The compressed data is then forwarded to the next VOL using `H5VLdataset_write()`. Reads are similar, except the compressed data is first read using the next vol with `H5VLdataset_read()` and then decompressed into `buf`.

Asynchronous VOL Operations

VOLs can internally spawn thread pools to perform asynchronous operations and remove overhead such as locking and I/O stalls from the critical path. HDF5 does not provide a thread pool API and expects users will use an external threading library, such as pthreads or Argobots. Many VOL functions such as `H5VLdataset_write()` are either synchronous or asynchronous depending on if an HDF5 request struct is passed to it. `H5VL_request_class_t` contains methods for waiting, canceling, completing, and freeing async requests. The VOL can define a custom request struct, taking as input the HDF request struct, and return it from VOL functions that behave asynchronously.

Introspecting VOLs

The introspection API provided by VOLs (`H5VL_introspect_class_t`) provides several abilities to query properties of the VOL. `get_conn_cls()` provides the ability to locate the class struct pointer for a VOL. The capability flags can be queried with `get_cap_flags`. Lastly, the `opt_query()` function

```c
typedef struct H5VL_compress_vol_info_t {
  int compress_method_;
  hid_t next_vol_id_;
  void *next_vol_info_;
} H5VL_compress_vol_info_t;
#define MAKE_TEMP_BUFFER \
  for (size_t i = 0; i < count; i++) { \
    next_objs[i] = objs[i]->next_vol_info_; \
    max_size[i] = get_data_size(mem_type_id[i], mem_space_id[i]); \
    compressed[i] = malloc(max_size[i]); \
  }
static herr_t H5VL_compress_vol_dataset_write(
    size_t count, void *dset[], hid_t mem_type_id[], hid_t mem_space_id[],
    hid_t file_space_id[], hid_t plist_id, const void *buf[], void **req) {
  H5VL_compress_vol_info_t **objs = (H5VL_compress_vol_info_t**)dset;
  void* next_objs[count]; void *compressed[count]; size_t max_size[count];
  MAKE_TEMP_BUFFER
  for (size_t i = 0; i < count; i++)
    compress(objs[i]->compress_method_, buf[i], max_size[i], compressed[i]);
  hid_t ret_value = H5VLdataset_write(
    count, (void**)next_objs, objs[0]->next_vol_id_, mem_type_id,
    mem_space_id, file_space_id, plist_id, compressed, req);
  // Free compressed & return
}
static herr_t H5VL_compress_vol_dataset_read(/** Similar to write */) {
  H5VL_compress_vol_info_t **objs = (H5VL_compress_vol_info_t**)dset;
  void* next_objs[count]; void *compressed[count]; size_t max_size[count];
  MAKE_TEMP_BUFFER
  hid_t ret_value = H5VLdataset_read(/** Same as write */);
  for (size_t i = 0; i < count; ++i)
    decompress(objs[i]->compress_method_, compressed[i], max_size[i], buf[i]);
  // Free compressed & return
}
```

Listing 4.2 – Compression pass-through VOL code example.

allows for determining whether or not optional operations are supported by this VOL connector, such as a particular method in the `H5VL_attr_class_t` struct. This is typically more fine-grained than the capability flag query. In addition, the `H5VL_info_class_t` provides the ability to query the VOL's specific info struct, if the VOL defines one. The function `from_str()` allows the info struct to be converted into a human-readable string.

Source File	Description
H5VLpublic.h	Public APIs for VOL registration and introspection
H5VLconnector.h	Public structs and APIs for VOL developers
H5VLconnector_passthru.h	Public APIs for pass-through VOL developers
H5VLpkg.h	Internal APIs for VOL registration
H5VLprivate.h	Internal APIs for VOL calls
H5VLmodule.h	Internal macros for VOLs and documentation
H5VL.c	Implementation of H5VLpublic.h
H5VLcallback.c	Implementation of H5VLconnector.h and H5VLconnector_passthru.h
H5VLint.c	Implementation of initialization functions in H5VLpkg.h
H5VLdyn_ops.c	Implementation of optional functions in H5VLpkg.h
H5VLtest.c	Internal helpers for environment variable checks

Source files for H5VL developers

4.2. Performance considerations

Scope.

- Skip list & hash table use

  The skip list is a simple in-memory data structure that allows O(log N) key/value insertion, lookup, and removal. Skip lists are used widely in the library for many tasks, including property list operations and I/O chunk selection (among many others). While skip lists should theoretically be efficient for these use cases as long as N does not grow extremely large, profiling has found that skip list operations are a frequent cause of library underperformance, even though the N values (number of properties, number of IDs) are not extremely large. We currently suspect this underperformance is due to the shape of the skip list data structures causing frequent CPU cache misses. More investigation is needed, and it may be possible to improve skip list performance with additional work. It is now recommended to avoid using the skip list in performance-critical sections of the code wherever possible. The primary motivation behind the context package (H5CX) is to avoid invoking skip list operations when handling property lists whenever possible. Unlike a classic skip list, HDF5's skip list is deterministic and does not use the random number generator and, therefore, will not interfere with the application's random numbers.

  In order to improve performance, we have more recently incorporated the open-source uthash [3] hash table into HDF5 and begun using it in places that previously would have used the skip list. A hash table offers, in general, O(1) insertion, lookup, and removal at the cost of higher memory usage. Therefore, a hash table can be expected to scale better than a skip list when handling large numbers of objects (again, at the cost of high memory usage). In practice, we have also found uthash to be more performant

even with smaller numbers of objects. We recommend that developers use uthash instead of skip lists whenever possible and whenever memory usage is not critical.

When objects inserted into these structures need to be iterated over in key order, the advantage of uthash is less apparent. In this case, items in the skip list are already in the correct order, but the uthash hash table must be sorted with `HASH_SORT`, which is an O(log N) operation. Likely, uthash is still faster in this case but needs more investigation.

- Allocations and deallocations

  In order to minimize the number of calls to system memory management functions, the HDF5 library has implemented the free list (`H5FL`) package. This package serves as a cache of allocated memory blocks and is effectively an intermediate memory manager between the library and the operating system. Regular free lists are the most commonly used type of free list and serve as a cache of allocations of a single structure type. Other free list types are sequence, array, and factory free lists that are used for variable-sized allocations and are less frequently used, and may not be as efficient as simply using the system memory manager. More investigation is needed.

  We have found mixed results when evaluating the performance of the free lists in HDF5. There is a configure option to disable free lists, sometimes leading to a greater performance with free lists disabled. It is likely that, when performing `calloc` operations, free lists are less efficient than simply using the system `calloc()` since the free list must `memset` the buffer, while the system can return a block from a page it knows has already been set to zero. For this reason, we recommend avoiding the use of `H5FK_CALLOC()` whenever possible, and when that is not possible, consider using the system `calloc()` instead, unless there is reason to believe the free list will be faster. Note that, since you cannot mix free list and system memory management calls on the same memory block, if `calloc` is only needed some of the time, it might require profiling to determine which method is best.

- Parallel performance considerations

  Parallel access can significantly improve raw data performance when HDF5 is used on a parallel file system. However, there are additional things to keep in mind to maximize performance and avoid falling into traps. For raw data, the most important decision to make is collective vs. independent I/O. While independent I/O gives more flexibility to the application, since it does not need to be called by every rank, collective I/O often gives better performance. This is because it allows MPI to intelligently distribute data in the most efficient manner possible while coordinating with other ranks. However, sometimes the overhead associated with collaboration, and the overhead associated with creating and unpacking the MPI datatypes, outweighs the performance benefits. It is difficult to tell which method will be faster without benchmarking, so it is a good idea to try independent, collective, and collective interface with independent low-level I/O (as set by `H5Pset_dxpl_mpio_collective_opt()`).

  The shape of the I/O can also have a substantial impact on raw data I/O performance. While the MPI datatype facility (used when called with the collective interface) should allow more efficient I/O with regularly strided selections because it allows such a pattern to be passed to MPI in a single I/O call, in practice it is still faster to use contiguous blocks for I/O. This often also extends even to very large blocks, where it is more efficient for each rank's chunks or blocks to be placed next to each other than to have the blocks from the different ranks interleaved.

  The application also has an option to use independent or collective I/O for metadata operations. Since all metadata write operations are required to be called collectively in HDF5, enabling metadata writes with `H5Pset_coll_metadata_write()` does not affect the semantics of the operation, and the

library still coordinates even if metadata writes are independent. Collective metadata writes often improve performance but it is worth checking to see which option works best for each application. In either case each piece of metadata is only written once.

Enabling metadata reads with `H5Pset_all_coll_metadata_ops()` changes the semantics of HDF5, requiring that all operations that read (or write) metadata be called collectively. This imposes an additional burden on the application, but in some cases can lead to substantial performance improvements. For example, if every process is opening the same dataset, then with the default independent metadata reads, every process issues its own read call and the filesystem is immediately swamped with N (number of ranks) read requests. However, when collective metadata reads are turned on, the filesystem only sees a single read request, with MPI internally distributing the data to all ranks in a much more efficient manner. These sorts of metadata read storms are a common cause of unexpected slowdowns in parallel code and should be among the first things to check for when diagnosing performance problems in parallel.

When creating new library code for parallel HDF5, developers must always consider what the most efficient use of MPI is. Coordination/communication is often much more expensive than computation on a single rank, so effort must be put in to minimize MPI calls that trigger network access, and to use the most efficient versions of these calls whenever possible. For example, don't call `MPI_Allgather()` when `MPI_Gather()` would suffice. In addition, it is often more efficient to compute the same values on every rank then to compute it once and broadcast the result. This can be seen in multiple places in the metadata cache and the I/O pipeline. Finally, the developer may need to balance memory usage against performance, since in some cases the memory available to each rank can be small (if there are many ranks per node) and things like MPI datatypes can use a substantial amount of memory. It may be necessary to add an API option to optimize for memory usage or performance, as is the case with link-chunk vs multi-chunk I/O.

4.3. HDF5 Library Extensions

4.3.1. Language Bindings

General Considerations

Language-binding interfaces serve the purpose of providing access to the C APIs and parameters to other programming languages. HDF5 offers three language wrappers for the C APIs: Fortran (Section 4.3.1), Java (Section 4.3.1), and C++. Each language component is developed separately from the C library component, and the build systems play a crucial role in obtaining information from the C library to ensure interoperability between the language interfaces.

The interface library aims to incorporate native language elements into the targeted language's API whenever possible. For example, the Fortran APIs avoid using `ISO_C_BINDING` KIND types such as `C_INT` and `C_FLOAT` in the API interfaces because most Fortran applications don't declare INTEGERs and REALs in the context of C interoperability. Instead, the Fortran wrappers handle the C interoperability at the wrapper level, making it unnecessary for a Fortran programmer to be familiar with C types in order to use the Fortran APIs.

The language bindings maintain their source, tests, and examples in isolation from the main C library tree. They can be enabled and disabled at build time without affecting the main C library. The testing schema for the language bindings has a different focus than the C library tests. Testing the language bindings focuses on their functionality and interoperability with the C APIs. In contrast, C testing aims to ensure the functionality of the C APIs in terms of the HDF5 library as a whole. Hence, the API wrapper testing, for the most part, assumes that the C API functionality has already been tested.

Fortran

The Fortran wrappers can be found in *fortran/src* in the HDF5 source code. The HDF5 modules topics are mapped to their corresponding *H5[**A**,**D**,**S**,**T**,**E**,**ES**,**F**,**Z**,**G**,**I**,**L**,**O**,**P**,**R**,**VL**]ff.F90* files, where ***A*** – Attribute, ***D*** – Dataset, ***S*** – Dataspace, ***T*** – Datatype, ***E*** – Error handing, ***ES*** – Event Set, ***F*** – Files, ***Z*** – Filters, ***G*** – Groups, ***I*** – Identifiers, ∅ – Library General, ***L*** –Links, ***O*** – Objects, ***P*** – Property List, ***R*** – References, ***VL*** –VOL connector. The older Fortran wrappers currently call a C wrapper to call the C APIs from. The C wrappers are located in *H5[**A**,**D**,**S**,**T**,**E**,**ES**,**F**,**Z**,**G**,**I**,**L**,**O**,**P**,**R**,**VL**]f.c*. Recent wrapper development, however, forgo calling intermediate C stubs and calls the C APIs directly from Fortran. The idea is to eliminate the necessity of the intermediate C wrappers, eventually using Fortran 2003 (F2003) and Fortran 2008 (F2008) features.

The F2003 and F2008 standards have simplified the interoperability between the C and the Fortran wrappers to a great extent. The Fortran wrappers take full advantage of this feature, which has improved their portability with the main C HDF5 library, reduced development time invested in Fortran, and made it easier to maintain them. These improvements were incorporated into the HDF5 library, starting with the 1.10 release. The following features of F2003 and F2008 are commonly used, and their availability is required for the wrappers:

- The `ISO_C_BINDING` and `ISO_FORTRAN_ENV` modules are used.
- C pointers are represented by an opaque derived type called `C_PTR`, and `C_FUNPTR` is used for callback functions.
- All Fortran APIs use the *BIND(C, name=**C API**)* convention, where ***C API*** matches the name of the C API.

With these new features, most, if not all, of the C wrappers can be eliminated by calling the HDF5 C APIs directly from the Fortran wrappers.

Lastly, the Fortran wrappers avoid modifying a user's buffer and try to minimize memory copies and allocations whenever possible. Consequently, when a C application reads data written from a Fortran program, the data appears to be transposed due to the difference in the storage order between C (row-major) and Fortran (column-major). For instance, if Fortran writes a 4x6 two-dimensional dataset to a file, a C program will read it into memory as a 6x4 two-dimensional dataset. It is worth noting that the HDF5 C utilities *h5dump* and *h5ls* display transposed data when data is written from a Fortran program.

An overview of the files, file generation, and other Fortran wrapper development practices are updated and detailed in *README.md*, located in *fortran/src*.

Java

The java wrappers can be found in `java/src` in the HDF5 source code. These wrappers are separated into two main components: the Java wrapper functions and the Java Native Interface (JNI) C functions.

The Java wrapper functions primarily consist of simple function declarations in `java/src/hdf/hdf5lib/H5.java` which contain the 'native' keyword, informing the Java Virtual Machine that those functions will be implemented by a matching JNI C function. Additional supporting Java infrastructure can be found in the following files/directories:

- `java/src/hdf/hdf5lib/H5.java`
  In addition to containing the Java wrapper function declarations, this file also contains code (in `loadH5Lib`) to load the C library that contains all of the JNI C functions that implement the Java wrapper functions. This code is run when the `H5` class is initialized and must successfully find and load the C library for the Java wrappers to operate correctly.
- `java/src/hdf/hdf5lib/HDF5Constants.java`
  This file defines a Java class that contains public variables for all of the constants that are available in the public C interface, usually as a #define macro, such as `H5F_LIBVER_LATEST`. Each variable has a name matching its corresponding macro in the C library, and the value for each variable is populated by having a similarly named JNI C function that simply returns the relevant value.
- `java/src/hdf/hdf5lib/HDFArray.java`
  This file defines functions to convert multi-dimensional arrays of bytes into arrays of Java objects and vice versa. Its main goal is to facilitate and simplify the I/O pathway between Java and C.
- `java/src/hdf/hdf5lib/HDFNativeData.java`
  This file defines Java wrapper functions for methods that support `HDFArray.java` during the data element conversion process.
- `java/src/hdf/hdf5lib/callbacks`
  This directory contains source files that define Java interfaces, functions, etc., necessary for being able to make a call from Java to an HDF5 API that accepts a pointer to a callback function, such as `H5Aiterate2` (which accepts an `H5A_operator2_t`).
- `java/src/hdf/hdf5lib/exceptions`
  This directory contains source files defining Java classes representing errors that can be returned from the HDF5 Java wrappers. The main error class defined is the `HDF5Exception` class, which extends Java's `RuntimeException`. Two other error classes are derived from this class, `HDF5LibraryException` and `HDF5JavaException`, from which several other error classes are derived. Errors derived from the `HDF5LibraryException` class correspond to errors that can be returned from the HDF5 API, while errors derived from the `HDF5JavaException` class corresponds to errors that occur in the Java wrapper functions and supporting infrastructure or in the JNI C code that implements the Java wrapper functions.
- `java/src/hdf/hdf5lib/structs`
  This directory contains source files that define Java classes which represent the C structures passed to certain HDF5 API calls, such as the `H5O_info_t` structure for `H5Oget_info`.

The Java wrapper functions currently do not handle any versioning of the HDF5 APIs and instead are typically updated to reflect the latest version of a versioned HDF5 API. For example, as of the time of writing the Java

wrapper for `H5Oget_info` is simply called `H5Oget_info`, but its parameters and the struct it returns (`H5O_info_t`) reflect the `H5Oget_info3` C API.

The JNI C functions reside in the source files under `java/src/jni`. In order for the Java Virtual Machine to be able to match a Java wrapper function with its C implementation, the JNI uses a name-mangling scheme somewhat similar to C++. The name for a function must follow the convention documented at Compiling, Loading and Linking Native Methods, where the name of the function is prepended with a prefix of "Java_", the fully-qualified Java method name (interspersed with underscores) and any needed escape sequences. For example, the JNI name for the `H5Dread` function is as follows:

```
Java_hdf_hdf5lib_H5_H5Dread
```

To conform to the specification that the JNI expects, each JNI C function should be marked with `JNIEXPORT` and `JNICALL` and must contain `JNIEnv *env, jclass clss` as the first two parameters, followed by the JNI equivalents for each of the parameters to the matching Java method. The parameters must also come in the same order as those in the matching Java method. For example, the full signature for the `H5Dread` function is:

```
JNIEXPORT jint JNICALL
Java_hdf_hdf5lib_H5_H5Dread(JNIEnv *env, jclass clss,
                            jlong dataset_id, jlong mem_type_id,
                            jlong mem_space_id, jlong file_space_id,
                            jlong xfer_plist_id, jbyteArray buf,
                            jboolean isCriticalPinning)
```

`JNIEnv *env` is a special parameter passed to each JNI C function by the JNI itself and is used to call C APIs that allow direct interaction with Java objects. `jclass clss` is also a special parameter passed to each JNI C function by the JNI but is mostly unused. For the other parameters, one can refer to JNI Types and Data Structures when translating between Java types and JNI types.

HDF5 JNI C functions are typically implemented as very thin wrappers around the HDF5 C API. Usually, simple argument checking is performed upon function entry, any necessary argument pinning may occur (described below), the relevant HDF5 C API call is made with the parameters passed to the wrapper function and then some cleanup is performed before returning from the function. Error handling in HDF5 JNI C functions are very similar in design to the HDF5 C library. Each function has a 'done:' label, and a set of macros is used to jump to this label with `goto` when an error occurs. Once an error occurs, these macros will construct an instance of the corresponding HDF5 Java exception class mentioned above and return that exception to the Java interface so it can be handled appropriately. If not handled at the Java interface level, the Java Virtual Machine may crash.

Due to the fairly large overhead incurred when crossing the boundary between Java and C code, there are a few best practices one should follow when designing new HDF5 JNI C functions:

- HDF5 Java wrapper functions should generally be designed such that the number of transitions between Java and C code to accomplish a task is minimized and as much work as possible is done within the C code portion of the wrapper function.
- To reduce the overhead of accessing Java array objects within C code, the JNI provides a process typically referred to as "pinning". This allows one to "pin" an array and get a direct pointer to the

elements of the array, rather than iterating through the array and accessing the elements individually with JNI calls. The `h5jni.h` header provides macros that wrap around the relevant JNI calls to pin arrays of each primitive datatype, such as `PIN_INT_ARRAY`, `PIN_INT_ARRAY_CRITICAL` and `UNPIN_INT_ARRAY`. These macros will return a pointer to the pinned array, as well as a boolean value that informs the caller whether the JNI had to make a copy of the array in order to pin it properly. If a copy had to be made, the caller **must** be sure to call the matching "unpin" macro (e.g., `UNPIN_INT_ARRAY`) before returning from the wrapper function. Otherwise, any changes to the array will not be reflected, and any resources used by the copied array will not be released. If performance is of the utmost importance in a wrapper function that deals with array objects, one may use the "critical" versions of these pinning macros rather than the regular versions. This approach will more likely give the caller an uncopied version of the array in question. Still, there are restrictions around when and how these particular macros can be used that the caller should be aware of (see GetPrimitiveArrayCritical, ReleasePrimitiveArrayCritical). Further information around this pinning process can be found at Accessing Primitive Arrays and Get<PrimitiveType>ArrayElements Routines.

A. Packages

HDF5 library functionality is divided into modules, which are implemented in packages. Packages are identified by a prefix of the form `H5X(Y)`. An example is the dataset package, which has the prefix `H5D`. In addition to the public APIs, many internal APIs are not visible via public API calls, like `H5FL` and `H5B2` (version 2 B-trees).

API calls, types, etc. in the HDF5 library have three *levels of visibility*. From most to least visible, these are:

- Public
- Private
- Package

Public things are in the public API. They are usually found in `H5*public.h` header files. API calls are of the form `H5*foo()`, with no underscores between the package name and the rest of the function name.

Private things are for use across the HDF5 library and can be used outside the packages that contain them. They collectively make up the “internal library API". They are usually found in `H5*private.h` header files. API calls are of the form `H5X_foo()` with one underscore between the package name and the rest of the function name.

Package things are for use inside the package and the compiler will complain if you include them outside of the package they belong to. They collectively make up the “internal package API." They are usually found in `H5*pkg.h` header files. API calls are of the form `H5X__foo()` with two underscores between the package name and the rest of the function name. The concept of "friend" packages exists, and you can declare this by defining `<package>_FRIEND` in a file. This will let you include the package header from a package in a file it is not a member of. Doing this is strongly discouraged, though. Test functions are often declared in package headers as they expose package internals, and test programs can include multiple package headers so they can check on the internal package state.

Note that the underscore scheme is primarily for API calls and does not extend to types and symbols. Another thing to remember is that the difference between package and private API calls can be somewhat arbitrary.

The current HDF5 library modules are listed in Tables A.1 and A.2.

As described in [8], HDF5 library packages contain functions with different scopes or visibilities for the API and other packages (public, private, package). The answer to the question of which package uses which other packages’ public functions can be found in Table A.3. A dot in a cell indicates that the package listed in the row heading uses private functions from the package listed in the column heading. For example, the dot (•) in the cell at position row `A` and column `I` indicates that the package `H5A` uses public functions from the package `H5I`.

H5* Package	Description
none (H5)	General use or library-wide routines
A	Attributes (including attribute I/O)
AC	Metadata cache (originally intended to be metadata-cache-specific routines)
B	V1 (legacy) B-tree index
B2	V2 B-tree index
C	Metadata Cache (originally intended to be generic cache code)
CS	Code stacks
CX	API call contexts
D	Datasets (including dataset I/O)
E	Error Handling
EA	Extensible array chunk index
ES	Event sets
F	HDF5 files
FA	Fixed array chunk index
FD	Virtual File Layer (VFL) and Virtual File Drivers (VFDs)
FL	Free lists
FO	Open file objects
FS	File free space tracking
G	Group objects and the HDF5 group structure
HF	Fractal heaps
HG	Global heaps
HL	Local heaps

Table A.1. – H5 – H5HL

H5* Package	Description
I	Identifiers (hid_ts)
L	Links
M	Map objects
MF	File memory management functions
MM	Memory management functions
O	Objects
P	Property lists
PB	Page buffering
PL	Plugins
R	Reference datatypes
RS	Reference-counted strings
S	Dataspaces
SL	Skip lists
SM	Shared object header messages
T	Datatypes
TS	Thread-safety
UC	Reference-counted objects
VL	Virtual Object Layer (VOL)
VM	Vector math routines
WB	Wrapped buffers
Z	Data filters and the data filter pipeline

Table A.2. – H5I – H5Z

Table A.3. – `H5*public` package dependencies (in HDF5 1.14.3 - will change over time)

→	A	AC	C	D	E	ES	F	FD	G	H5	I	L	M	MM	O	P	PL	R	S	T	VL	Z
A	○									•	•				•					•		
AC		○	•							•												
C			○							•												
D				○						•	•											
E					○					•	•											
ES						○				•	•											
F		•					○			•	•											
FD							•	○		•	•											
G									○	•	•	•			•							
H5										○												
I										•	○											
L										•	•	○			•					•		
M										•	•		○								•	
MM										•				○								
O										•	•				○							
P		•		•			•	•		•	•	•		•	•	○			•	•		•
PL										•							○					
R									•	•	•				•			○				
S										•	•								○			
T										•	•									○		
VL										•	•										○	
Z										•												○

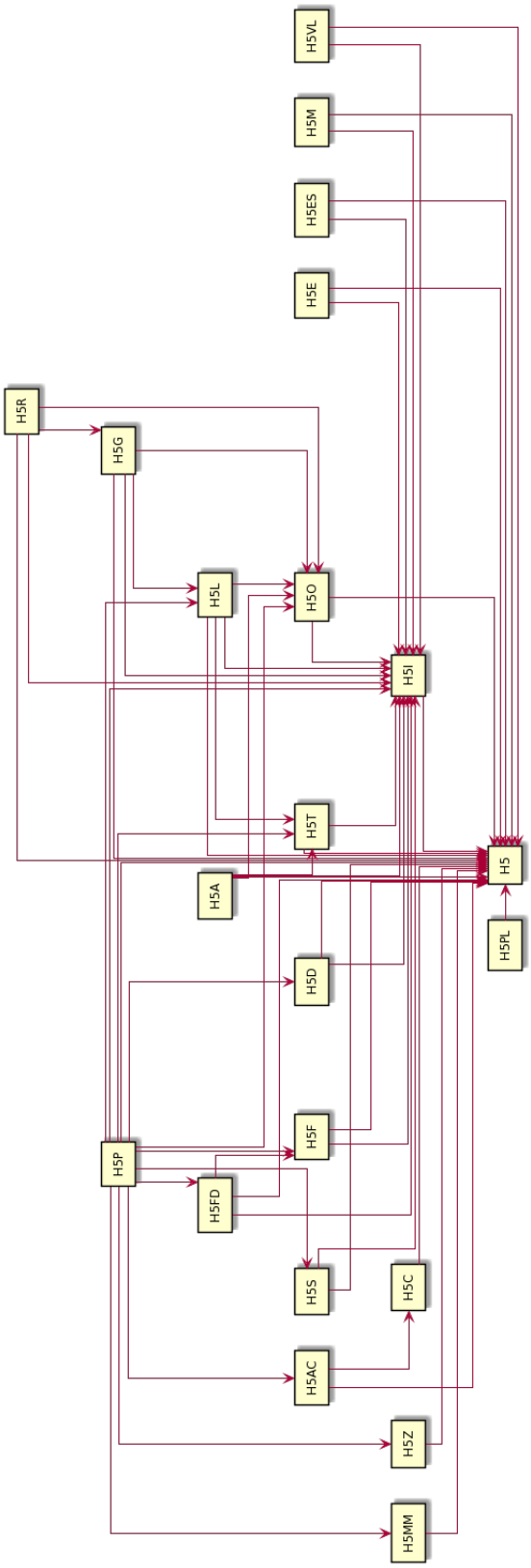


Figure A.1. – The public dependence of HDF5 library modules as a directed graph.

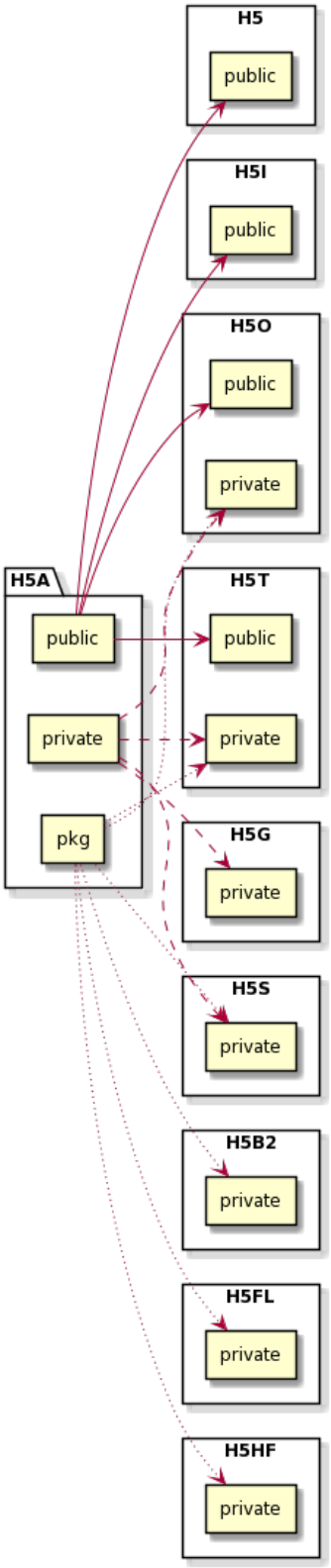


Figure A.2. – `H5A` package dependence across visibilities.

B. Feature chronology

Version	Major Feature Introduction
1.8.0	External links
	Non-persistent free-space tracking
	Unlimited attribute size
1.8.5	Group compression
1.10.0	Single-Writer Multiple Reader (SWMR)
	Virtual Datasets (VDS)
	64-bit `hid_t`
1.10.1	Paged allocation and persistent free space tracking
1.10.2	Parallel compression
1.12.0	Virtual Object Layer (VOL)
1.14.0	Multi-dataset I/O
	Selection I/O
	Overhaul of VOL
	Subfiling VFD

C. Feature (in-)compatibility

C.1. Operating Systems

C.1.1. Windows

- SWMR should work on Windows with local file systems (NOT SMB shares), though testing is not a robust as on MacOS/Linux
- Parallel HDF5 is poorly tested on Windows
- The direct VFD is not supported as it uses the POSIX `O_DIRECT` flag
- Thread-safety uses win32 threads
- Some API calls use `off_t`, which is a 32-bit type on Windows (we use `__int64` internally)
- Unicode file paths are not well-supported

C.2. Build Systems

All features should be supported in both the Autotools and CMake.

C.3. Thread-Safety

Thread-safety is currently implemented via a global API lock, which works with any API call in the HDF5 C library. This includes parallel HDF5, even though the build system will disallow this combination in earlier versions of the library (for no particular reason other than we didn't test it).

The global lock is not implemented in any of the wrapper or higher-level libraries, so they are NOT thread-safe:

- High-Level Library
- C++ API
- Fortran API
- Java API

C.4. Feature Compatibility

Feature	Serial	SWMR (serial)	Parallel
Asynchronous Operations	1	1	1
Compression (Filtering)	Y	Y	11
Direct Chunk I/O	Y	Y	8 (?)
Direct VFD	2	2	2
External Data Storage	Y	Y	9
External Links	Y	7	Y
Evict-on-Close	Y	Y	N
File Image	Y	N	N
Free-Space Tracking	Y	N	Y
Metadata Cache Image	Y	Y	Y
Metadata Page Buffering	Y	N	N
Metadata Updates	Y	5	6
Multi-Dataset I/O	3	3	3
Selection I/O	3	3	3
User-Defined Links	Y	7	Y
Variable-Length Datatypes	Y	N	7
Vector I/O	3	3	3
Virtual Dataset (VDS) Layout	4	4	10

1. Requires a suitable VOL connector (NOT the native connector)
2. Requires the POSIX `open(2)` call to accept the `O_DIRECT` flag
3. Requires a suitable virtual file driver (e.g., MPI-I/O) to see a benefit
4. Does not support point selections
5. Extending datasets only
6. Requires collective metadata operations if writing to a common file
7. Read-only
8. Not with filtered chunks
9. Independent only
10. Parallel VDS has limited functionality
11. Limited to early allocation

VFD stackability and features

Pass-Through	family, mirror, multi/split, onion, splitter, subfiling
Terminal	core, direct, HDFS, Hermes, log, MPI-I/O, sec2, ros3, stdio

NOTE: The onion and subfiling VFDs are currently hard-coded to use the sec2 VFD.

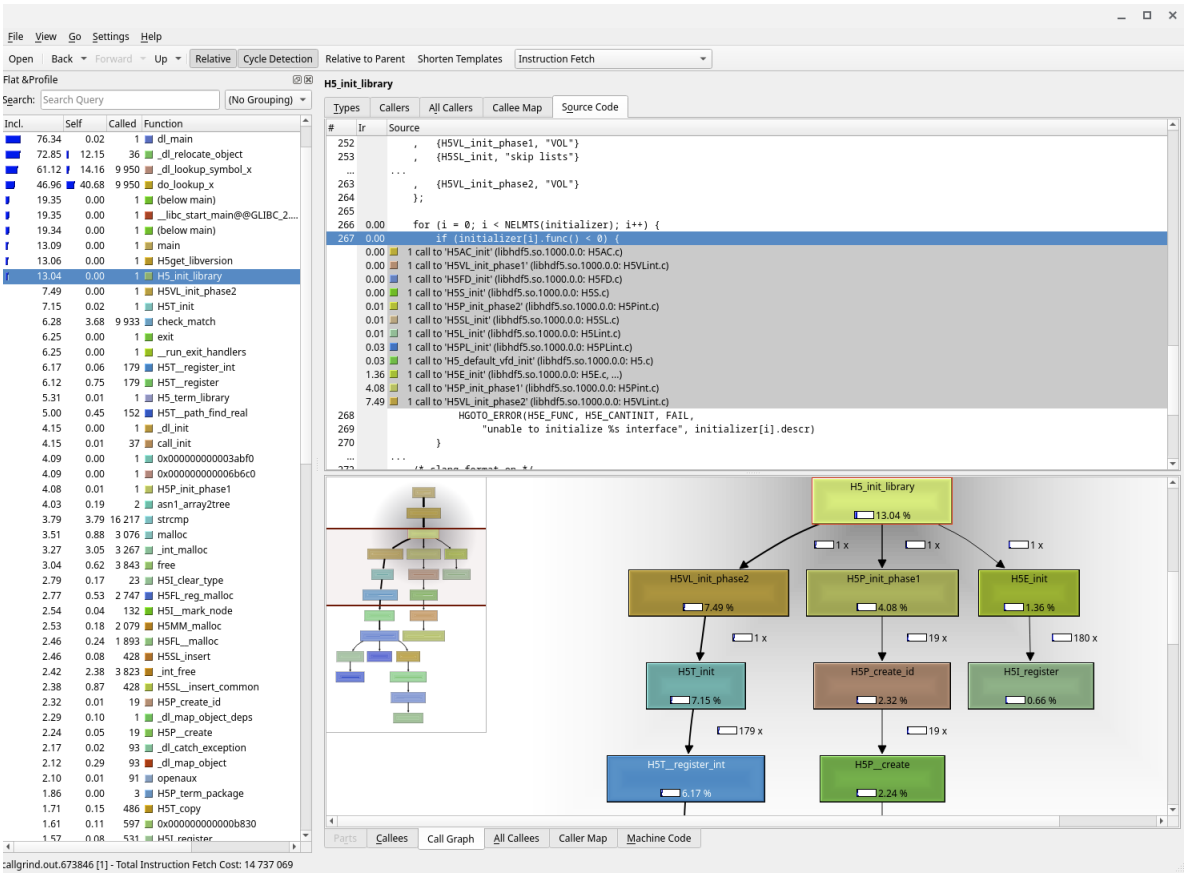
D. Environment

What you need to follow along:

- HDF5 library source code [11]
- GCC [5] (compile stuff w/ `-g`)
- Valgrind [4]
- KCachegrind [23]
- CMake [13]
- A clone of the GitHub repository [9], which contains the code samples

```
# run the program with callgrind; generates a file callgrind.out.XYZUV
valgrind --tool=callgrind ./executable

# open profile.callgrind with kcachegrind
kcachegrind callgrind.out.XYZUV
```



Ease of navigation. . . , but don't be fooled by pictures! You must look at the code.

Figure D.1. – Caption

```
...
struct {
    herr_t (*func)(void);
    const char *descr;
} initializer[] = {
    {H5E_init, "error"}
,   {H5VL_init_phase1, "VOL"}
,   {H5SL_init, "skip lists"}
,   {H5FD_init, "VFD"}
,   {H5_default_vfd_init, "default VFD"}
,   {H5P_init_phase1, "property list"}
,   {H5AC_init, "metadata caching"}
,   {H5L_init, "link"}
,   {H5S_init, "dataspace"}
,   {H5PL_init, "plugins"}
/* Finish initializing interfaces that depend on the interfaces above */
,   {H5P_init_phase2, "property list"}
,   {H5VL_init_phase2, "VOL"}
};

for (i = 0; i < NELMTS(initializer); i++) {
    if (initializer[i].func() < 0) {
        HGOTO_ERROR(H5E_FUNC, H5E_CANTINIT, FAIL,
            "unable to initialize %s interface", initializer[i].descr)
    }
}
...
```

Bibliography

[1] Mohamad Chaarawi. *RFC: Page Buffering*. URL: https://docs.hdfgroup.org/hdf5/rfc/RFC-Page_Buffering.pdf. (accessed: 14.03.2023).

[2] Vailin Choi, Quincey Koziol, and John Mainzer. *RFC: HDF5 File Space Management: Paged Aggregation*. URL: https://docs.hdfgroup.org/hdf5/rfc/paged_aggregation.pdf. (accessed: 14.03.2023).

[3] The UTHASH Developers. *UTHASH Github*. URL: https://github.com/troydhanson/uthash. Accessed: 15.12.2023.

[4] The Valgrind Developers. *Valgrind Home*. URL: https://valgrind.org/. (accessed: 21.06.2023).

[5] Free Software Foundation. *GCC, the GNU Compiler Collection*. URL: https://gcc.gnu.org/. (accessed: 08.03.2023).

[6] Free Software Foundation. *GNU Autotools*. URL: https://en.wikipedia.org/wiki/GNU_Autotools. (accessed: 05.07.2023).

[7] Free Software Foundation. *What is Free Software?* URL: https://www.gnu.org/philosophy/free-sw.en.html. (accessed: 08.03.2023).

[8] The HDF Group. *Getting Started with HDF5 Development*. URL: https://github.com/HDFGroup/hdf5/blob/develop/doc/getting-started-with-hdf5-development.md. (accessed: 03.08.2023).

[9] The HDF Group. *GitHub repository of examples*. URL: https://github.com/HDFGroup/arch-doc. (accessed: 01.09.2023).

[10] The HDF Group. *HDF5 File Format Specification Version 3.0*. URL: https://docs.hdfgroup.org/hdf5/develop/_f_m_t3.html. (accessed: 04.04.2023).

[11] The HDF Group. *HDF5 library and file format*. URL: https://github.com/HDFGroup/hdf5. (accessed: 08.03.2023).

[12] Chris Hanson and Gerald Jay Sussman. *Software design for flexibility: how to avoid programming yourself into a corner*. MIT Press, 2021.

[13] Inc. Kitware. *CMake*. URL: https://cmake.org/. (accessed: 05.07.2023).

[14] Jeffery A Kuehn. "Faster Libraries for Creating Network-Portable Self-Describing Datasets". In: *Proceedings of the 37th Cray User Group Meeting, Cray User Group, Inc., Barcelona, Spain, March*. 1996.

[15] Chee Wai Lee. *HDF5 Thread Safe library*. URL: https://docs.hdfgroup.org/archive/support/HDF5/doc/TechNotes/ThreadSafeLibrary.html. (accessed: 06.07.2023).

[16] Mark W Maier. *The art of systems architecting*. CRC press, 2009.

[17] OpenAI. *ChatGPT*. Accessed: 21.06.2023. 2023. URL: https://chat.openai.com/?model=gpt-4.

[18] The Linux Documentation Project. *10.1 An Overview of TCP/IP Networking*. URL: https://tldp.org/LDP/tlk/net/net.html. (accessed: 05.07.2023).

[19] Jerome Soumagne, Quincey Koziol, and Pourmal Elena. *RFC: New Datatypes*. URL: https://docs.hdfgroup.org/hdf5/rfc/new_datatypes.pdf. (accessed: 14.03.2023).

[20] Jerome Soumagne et al. "Accelerating HDF5 I/O for Exascale Using DAOS". In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 903–914.

[21] Houjun Tang et al. "Enabling Transparent Asynchronous I/O using Background Threads". In: *Proceedings of 2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*. 2019.

[22] R. Thakur, W. Gropp, and E. Lusk. "Data sieving and collective I/O in ROMIO". In: *Proceedings. Frontiers '99. Seventh Symposium on the Frontiers of Massively Parallel Computation*. 1999, pp. 182–189. DOI: 10.1109/FMPC.1999.750599.

[23] Josef Weidendorfer. *KCachegrind*. URL: https://kcachegrind.github.io/html/Home.html. (accessed: 21.06.2023).

[24] David D. Woods. *The theory of graceful extensibility: Basic rules that govern adaptive systems - environment systems and decisions*. Sept. 2018. URL: https://link.springer.com/article/10.1007/s10669-018-9708-3.