Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELORTHESIS

Database VOL-plugin for HDF5

vorgelegt von

Olga Perevalova

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Wirtschaftsinformatik
Matrikelnummer:	6525419
Erstgutachter:	Dr. Michael Kuhn
Zweitgutachter:	Prof. Dr. Thomas Ludwig
Betreuer:	Eugen Betke

Hamburg, 2017-07-05

Abstract

HDF5 is an open source, hierarchical, and self-describing format for flexible and efficient I/O for high volume and complex data, that combines data and metadata. Advantages of this format make it widely used by many scientific applications.

In a parallel HDF5 application when a large number of processes access a shared file simultaneously synchronization mechanism used by many file systems may significantly degrade I/O performance. Separation of metadata and data is the first step to solve this problem.

The main contribution of this thesis is a prototype of an HDF5-VOL-Plugin that separates metadata and data. To this end, metadata are stored in an SQLite3 database and data in a shared file. It uses MPI for synchronization of metadata when several processes access the SQLite3 database.

In the context of this work a benchmark test has been developed. It measures access times for each metadata operation and the overall I/O performance. The execution time of the Database VOL-plugin is compared to the native solution.

The test results show that the database plugin consistently demonstrates good performance.

The thesis concludes with a critical discussion of the approach by looking at the metadata from different perspectives: scientific applications vs. HDF5.

Contents

1 Introduction **5**
- 1.1 Self-Describing Data Formats . . . 5
- 1.2 Hierarchical Data Format 5 (HDF5) . . . 6
- 1.3 HDF5 Virtual Object Layer (VOL) . . . 7
- 1.4 Motivation . . . 8
- 1.5 Contribution . . . 9
- 1.6 Structure . . . 9

2 Technical Background **10**
- 2.1 The HDF5 Data Model . . . 10
  - 2.1.1 The HDF5 File . . . 11
  - 2.1.2 The HDF5 Group . . . 11
  - 2.1.3 The HDF5 Dataset . . . 12
  - 2.1.4 The HDF5 Dataspace and Datatype . . . 12
  - 2.1.5 The HDF5 Attribute . . . 13
- 2.2 The HDF5 Programming Model . . . 14
  - 2.2.1 The HDF5 Library . . . 14
  - 2.2.2 Working with an HDF5 File . . . 15
  - 2.2.3 Working with an HDF5 Group . . . 16
  - 2.2.4 Working with an HDF5 Dataset and with an HDF5 Attribute . . . 17
- 2.3 The Database VOL-Plugin for HDF5 . . . 18
  - 2.3.1 VOL Structure . . . 18
  - 2.3.2 Mapping the API to the Callbacks . . . 20

3 Related Work **23**
- 3.1 Separation of metadata and data for other file formats . . . 23
- 3.2 Storage of HDF5 Objects in separate locations . . . 23

4 Design **25**
- 4.1 Model of the Database VOL Plugin . . . 25
- 4.2 Storage of HDF5 Metadata in a Relational Database . . . 26
  - 4.2.1 Representation of HDF5 Objects . . . 26
  - 4.2.2 Working with the Database VOL Plugin . . . 29

5 Implementation **32**
- 5.1 The Main Components . . . 32
- 5.2 Create/Open/Close the Database and the Data File . . . 33

5.3 Dataset Create Routine . . . 33
5.4 Dataset Open Routine . . . 36
5.5 Dataset Get Routine . . . 36
5.6 Dataset Read/Write Routine . . . 37
5.7 Attribute Read/Write Routine . . . 39
5.8 The barrier synchronization . . . 39

6 Evaluation **42**
6.1 Test Environment . . . 42
6.2 Test Configurations . . . 42
6.3 Test Results and Evaluation . . . 44
6.3.1 Metadata performance . . . 44
6.3.2 Data performance . . . 47
6.3.3 Metadata: HDF5 vs. scientific applications . . . 48

7 Conclusion and Future Work **51**
7.1 Conclusion . . . 51
7.2 Future Work . . . 51

Bibliography **52**

List of Figures **54**

List of Listings **55**

List of Tables **56**

1 Introduction

In this chapter, basic information about a popular data format library HDF5 that is used by many scientific applications will be introduced; it will be also briefly discussed an abstraction layer internal to the HDF5 data model called the Virtual Object Layer (VOL). A special focus lies on the HDF5 problems with data storage performance for small and non-contiguous disk accesses.

1.1 Self-Describing Data Formats

Self-describing data format allows an application to interpret the structure and contents of a file with no outside information.

The following advantages are the main reasons why self-describing data formats became widely used by many scientific applications [CT]:

- Before standard file formats were developed, custom file formats were used in many projects. Often they were raw binary or ASCII[1], which caused some problems.
  - Machine dependent byte ordering or floating-point organizations required a key in order to open the file and read the right data.
  - A new custom reader is needed for each different data organization.
- Self-describing data formats include a standard API and portable data access libraries.
- Self-describing data formats include tools to open and work with files, using the embedded descriptions to interpret the data.

Additionally, the use of self-describing data formats has proven valuable in archiving and disseminating scientific data. For example, Hierarchical Data Format (HDF) and Network Common Data Form (NetCDF) are two self-describing formats, that are widely used by agencies including NASA[2] and NOAA[3] [CT].

[1] American standard code for information interchange
[2] National Aeronautics and Space Administration
[3] National Oceanic and Atmospheric Administration

1.2 Hierarchical Data Format 5 (HDF5)

HDF5 is an open source, hierarchical, and self-describing format which combines data and metadata. HDF5 is maintained by a non-profit corporation, The HDF Group, whose purpose is to ensure the sustainable development of HDF5 and continued accessibility of HDF-stored data.

HDF5 implements a model for managing and storing high volume data. The model consists of [Gro16]:

- Open file format - an abstract storage model for storing large amounts of data.
- Data model - structures for logically organizing and accessing HDF5 data from an application (an abstract data model).
- Open source software - libraries to implement the abstract model and to map the storage model to different storage mechanisms, language interfaces and tools for working with data in this format.

Figure 1.1 illustrates the HDF5 model and its implementations [Gro15a].

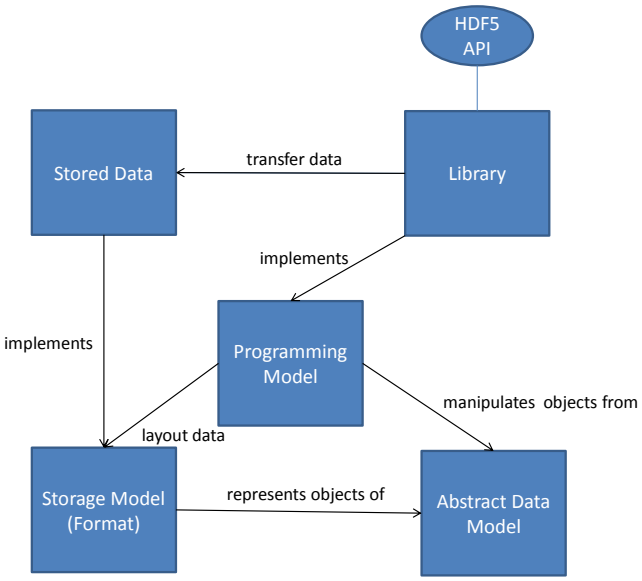

Figure 1.1: HDF5 Models and Implementations[Gro15a]

HDF5 has an object oriented ideas for storing data. Abstractly, an HDF5 file is a container that holds data objects such as images, tables, graphs, documents. HDF5 objects can be thought of as building blocks for data organization and specification [QK14].

The HDF5 data model is composed of two basic objects, groups and dataset: an HDF5 file contains a hierarchy of numerical arrays of data elements (datasets) organized within groups. There are also a variety of other objects in HDF5: datatypes, dataspaces, properties, attributes etc.

1.3 HDF5 Virtual Object Layer (VOL)

The VOL is an abstraction layer in the HDF5 library. Its purpose is to expose HDF5 API to applications and provide a simple way of using different storage mechanisms. VOL is right below the public HDF5 API, it intercepts all API routines that could potentially modify the HDF5 file and forwards those calls to plug-in object drivers (see Figure 1.2) [Cha14b] [Gro15b].

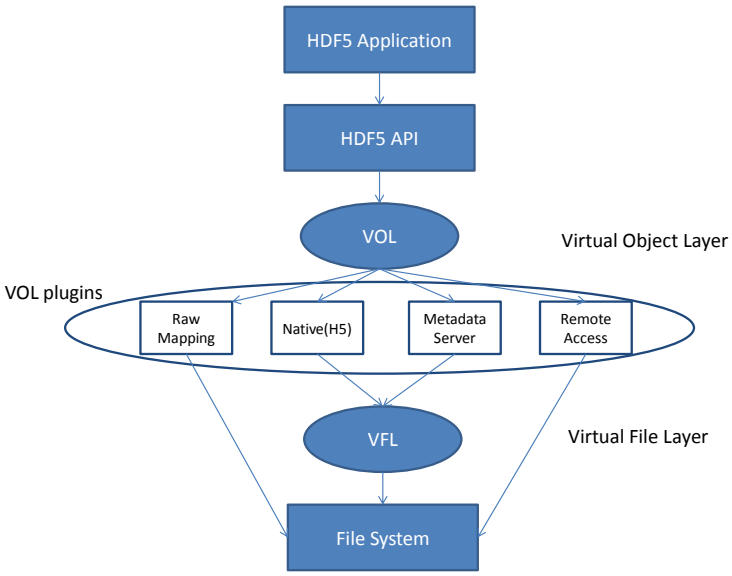

Figure 1.2: Virtual Object Layer(VOL)[Gro15b]

VOL provides a flexible way to write and read data. For example, a plugin could distribute data objects remotely over different platforms or store the data in other file formats. The VOL maintains the HDF5 object model and allows writing plugins for accessing HDF5 data [Cha14b].

1.4 Motivation

It is very important to maintain the correct relationship between mass storage devices costs and their performing efficientcy.

In most memory-consuming research areas like climate research, where petabytes of data are collected and analyzed, mass storage costs are significant compared to the overall system costs. Because of the cost, mass storage devices in such areas have tended to be much larger and at the same time much slower. There are some reasons why separation of metadata and data would be reasonable.

Use case 1: In a large-scale application, due to the domain decomposition, in most cases, individual processes do not need complete metadata. Therefore, they request only the parts of them, that are required for the processing of data. Since the most I/O middleware and network do not guarantee the arrival of I/O requests in sequential order, and the application processes run at different speed, it is most likely that they arrive at the I/O nodes in more or less random order. Considering that large-scale applications access relative small metadata with a high number of processes, it results in a large number of small and non-contiguous disk accesses and probably in a poor performance. For example, a user who runs parallel jobs on 1000 processing nodes and 24 processed per node needs to access HDF5 data. In that case, supporting metadata would be accessed by all these processes (24000) nearly at the same time. In an HD(CP) experiment [EB], when accessing an float-array with 22282304 elements it would results in 24000 of 3.6 KiB random accesses. Metadata consists of 21 of such datasets would increase the number of accesses to 504000.

Use case 2: Typically, the size of supporting metadata is small compared to data; therefore, it could be reasonable to store metadata separately on a fast and highly available storage device and data on a slow, but cheap storage (for example, tape). On a such system, it is possible to implement a search service, that can use metadata, to find required datasets.

Also, some small data or data in a coarser granularity as a preview can be stored. It makes it possible search services to implement, which can also for data in the archive look. Such service would not use slow tape storage; therefore, it would need relatively not much time for searching in the whole archive. For example, a user can for particular region search, using longitude and latitude or for simulation data using time interval. Besides saving of time, external catalogue management would be not needed.

In addition, the application can work separately with metadata and does not need to read the data into memory, which reduces the I/O time.

Use case 3: Separation of metadata and data can be used as the basis for the further development of a high-performance HDF5 plugin. The parallelization of data I/O is much easier, when metadata are stored separately.

1.5 Contribution

The purpose of this thesis is to develop an external plugin implemented using HDF5 VOL that stores data and metadata to different locations. Metadata to a database and data to file system.

The key contributions of this thesis include:

- Design of the database framework for the HDF5 metadata.
- Design and implementation of the Database VOL Plugin for HDF5.
- Evaluation of this plugin.

1.6 Structure

This rest of the thesis is organized as follows: Chapter 2 introduces the HDF5 data model and a programming model that manipulates objects from the data model. It also discusses the implementation of the Database VOL Plugin for HDF5: the callback interface defined for the VOL and mapping the HDF5 API calls to the callbacks. Related works are reviewed in chapter 3. The design of the Database VOL Plugin is described in Chapter 4. Chapter 5 presents some parts of the implementation. Chapter 6 gives an overview of the experiments and performance results. Chapter 7 provides the conclusion.

Summary

This chapter has briefly introduced data format library HDF5 and the HDF5 Virtual Object Layer(VOL). The VOL exports an interface that allows writing plugins for accessing HDF5 data. The Database VOL Plugin would allow storing data and metadata separately. Better HDF5 data storage performance for small and non-contiguous disk accesses could be reached with the use of the plugin.

2 Technical Background

In this chapter, the HDF5 abstract data model and the HDF5 programming model will be introduced in detail. This chapter also will provide the information related to the implementation of the VOL Plugins for HDF5: the callback interface defined for the VOL and mapping the HDF5 API calls to the callbacks.

2.1 The HDF5 Data Model

An abstract data model organizes data, data types, and standardizes how they are related to one another.

Table 2.1 presents objects defined with the HDF5 data model [Gro15a] .

Object	Description
File	A container for organized HDF5 objects
Group	An organizational structure containing zero or more groups or datasets, with supporting attributes and other metadata
Dataset	A numerical array of data elements with attributes and other metadata
Dataspace	A description of organisation of the data elements
Datatype	A description of the format of a single data element
Attribute	A small metadata object associated with a group, dataset, or named datatype
Property List	A collection of parameters controlling options in the library
Link	The way objects are connected.

Table 2.1: The HDF5 Objects

Some objects of the HDF5 abstract data model are of special interest for this thesis; therefore, they are presented in detail below.

2.1.1 The HDF5 File

Formally, an HDF5 file is a container for organized groups, datasets, and other HDF5 objects [Gro16]. The objects are organized as a rooted, directed graph. Every HDF5 file has at least one object, the root group. Figure 2.1 shows an HDF5 file structure with groups and datasets.

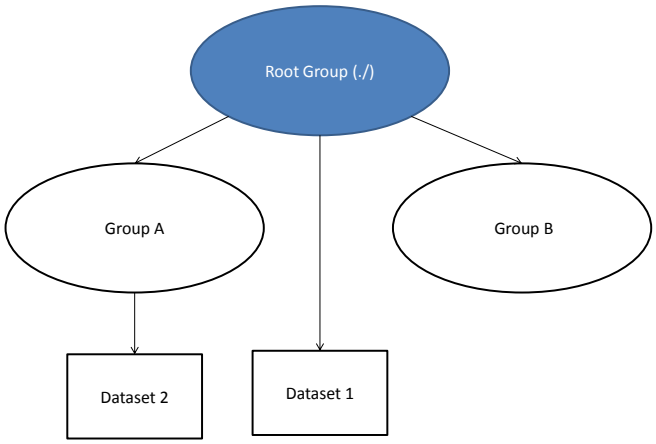

Figure 2.1: A HDF5 file structure with groups and datasets

2.1.2 The HDF5 Group

The HDF5 groups are primary HDF5 objects. They organize data objects; therefore, groups can be used to implement the hierarchy of HDF5.

A group can contain zero or more objects. Each HDF5 object must be a member of at least one group. The root group is a special group with the name /. A HDF5 file should have the root group: the root group is automatically created when the HDF5 file is created [Gro15a]; therefore, other HDF5 objects are members or descendants of the root group.

Working with HDF5 groups is analogous to working with a file system directory. Directories (or folders) in a file system allow the user to group files into separate collections and files are defined by their path names. HDF5 groups are similar to the directories and folders; HDF5 datasets are similar to the files. As with a file system, objects in HDF5 have a unique identity within the hierarchy of the file and can be accessed by giving their path names.

But there are some important differences between a file system directory and a HDF5 hierarchy. Groups in HDF5 allow circular references, whereas the file systems are strictly

hierarchical, allowing no circular references [Gro15a]. For example, Group B is a member of Group A, and Group C is a member of Group B. In a case of circular references, Group A can be a member of Group C. Secondly, it is possible to add metadata to groups, whereas file systems do not support this.

2.1.3 The HDF5 Dataset

The HDF5 datasets are multidimensional arrays, that organize and contain data elements (sets of bits). Besides data, a dataset consists of supporting metadata that stores a description of the data elements, data layout, and all other information necessary to write, read, and interpret the stored data [Gro15a].

The number of dimensions, size of each dimension are described by the dataspace object and the layout of the bits of a data element is described by the datatype. These are independent objects and are created separately from any dataset to which they may be attached. The dataset object also may have several attributes. The HDF5 dataspace, datatype und attribute are described in the next sections.

Figure 2.2 illustrates an application view of the dataset. [Gro15a].

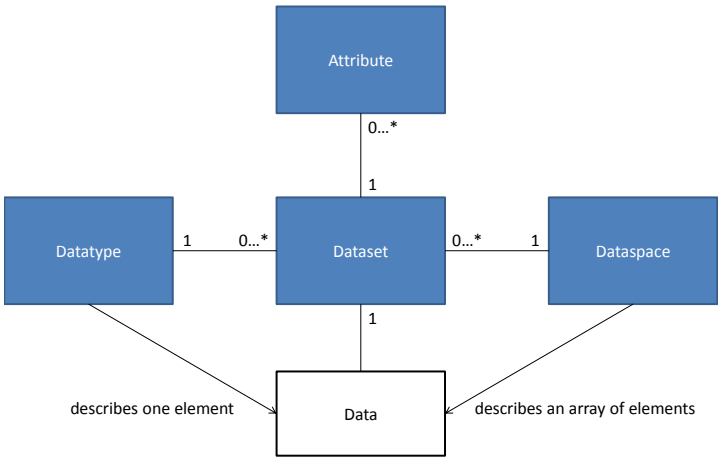

Figure 2.2: Application view of the HDF5 dataset

2.1.4 The HDF5 Dataspace and Datatype

The HDF5 datatypes and dataspaces are required components of an HDF5 dataset or attribute definition [Gro15a]:

- A dataspace object describes the number of dimensions and the size of each dimension of the multidimensional array in which the data are represented.

- A datatype object describes the individual data element (one specific layout of bits) and provides complete information for data conversion to or from that datatype. The datatype applies to every data element and cannot be changed.

HDF5 supports a wide variety of datatypes [Gro16] (see Table 2.2).

Datatype	Description
Integer	Twos complement integers
Float	Floating point numbers
Character	Array of 1-byte character encoding
Bitfield	String of bits
Opaque	Uninterpreted data
Enumeration	A list of discrete values, with symbolic names in the form of strings
Reference	A reference to object or region within the HDF5 file
Array	An array of data elements
Variable-length	A variable-length 1-dimensional array of data elements
Compound	A datatype of a sequence of datatypes

Table 2.2: The HDF5 Datatypes

2.1.5 The HDF5 Attribute

An HDF5 attribute is a small metadata object, that can optionally be associated with a group, dataset, or named datatype. These objects may have zero or more user defined attributes.

HDF5 attributes consist of a name and data and look very similar to HDF5 datasets in that they have a datatype and a dataspace.

But there are some differences [Gro15a]:

- An attribute should be a small object.
- An attribute does not have attributes.
- An attribute can only be accessed via the object it is attached to.
- There is no partial reading or writing.

2.2 The HDF5 Programming Model

A programming model manipulates objects from an abstract data model. The programming model of HDF5 is just as important as the abstract data model, and the principles used in both of them should be applied to design of the Database VOL Plugin.

The general paradigm for working with HDF5 objects is to:

- Create or open the object.
- Perform the desired operations on the object.
- Close the object.

2.2.1 The HDF5 Library

The HDF5 library implements the programming model; therefore, it should be considered in this chapter.

The library is implemented in C programming language. Although it does not support OOP[1], the library uses several mechanisms to perform an object oriented ideas for storing data [Gro15a].

The library implements HDF5 objects as data structures. There are special pointers - identifiers; an identifier is used as a reference to a specific instance of an object. For example, when a HDF5 file is created, the API returns a file identifier. The identifier references all operations on that file and remains valid until the file is closed [Grod].

In OOP all available procedures for a given object are called methods of that object. HDF5 library simulates object methods through its API naming convention.

The table below presents the HDF5 objects and the standard prefixes indicating the type of object on which the function operates [Gro16](see Table 2.3).

Prefix	The type of object
H5A	Attribute
H5D	Dataset
H5F	File
H5G	Group
H5L	Link
H5O	Object
H5P	Property List
H5S	Dataspace
H5T	Datatype

Table 2.3: The HDF5 API naming scheme

[1]Object-Oriented Programming

2.2.2 Working with an HDF5 File

Before objects in HDF5 can be used or referred, they must be explicitly created or opened. When the file is created or opened, the file access modes and the file creation properties or file access properties specify permissions and settings for the file [Grob].

File Modes

Two modes can be used with a create function and two with an open function (see Table 2.4) [Grob].

Access Function	Access Flag	Access Mode
H5Fcreate	`H5F_ACC_EXCL`	If the file exists, create function fails.
H5Fcreate	`H5F_ACC_TRUNC`	If the file exists, create function overwrite it.
H5Fopen	`H5F_ACC_RDONLY`	If the file does not exist, open function fails; otherwise, read-only access.
H5Fopen	`H5F_ACC_RDWR`	If the file does not exist, open function fails; otherwise, read-write access.

Table 2.4: File Access Modes

In the example below, when a new file is created but a file of the same name already exists, the access flag `H5F_ACC_TRUNC` allows overwrite any existing data(see Listing 2.1).

```
1 file = H5Fcreate
    ↪ ("File.h5",H5F_ACC_TRUNC,H5P_DEFAULT,H5P_DEFAULT);
```

Listing 2.1: Creating an HDF5 file using read-write modus

File Properties

Defaults properties (`H5P_DEFAULT`) handle the most common needs [Grob]. File creation and file access properties control the more complex aspects of creating and accessing files[Gro15a]. The file creation properties specify version information and parameters of global data structures; they are set permanently for the life of the file [Gro15a]. File creation properties control the following characteristics[Grof]:

- The size of the user-block (a fixed length block of data located at the beginning of the file which may be used to store any data information found to be useful to applications).

- The number of bytes that are used to store the offset and length of objects in the file.
- Properties of the B-trees that are used to manage the data in the file.
- Library versioning information.

The file access properties are used to control different methods of performing access operations on files; they can be changed by closing and reopening the file [Gro15a]. Some of them are [Grof]:

- Properties for data alignment.
- Properties for parallel access to a file through the MPI I/O library.
- Properties for access to local temporary files directly from memory without ever creating permanent storage.

This following example shows how to initialize the file for parallel access [KA10].

```
1 /* create access property list */
2 plist_id = H5Pcreate(H5P_FILE_ACCESS);
3 /* necessary for parallel access */
4 status = H5Pset_fapl_mpio(plist_id, MPI_COMM_WORLD,
    ↪ MPI_INFO_NULL);
5 /* Create an hdf5 file */
6 file_id = H5Fcreate(FILENAME, H5F_ACC_TRUNC, H5P_DEFAULT,
    ↪ plist_id);
7 status = H5Pclose(plist_id);
```

Listing 2.2: Initialization the file for parallel access

For more information on the properties for other HDF5 objects refer to the HDF5 user guide.

2.2.3 Working with an HDF5 Group

Before a group can be created or opened, the location identifier must be obtained. For example, in order to create the group in the file (root group), the function `H5Gcreate` must be called, passing identifier returned from opening or creating the file.

A group may be created within another group by providing an absolute path to the function or by specifying its location. The full name (the absolute path) would include a tracing of the group hierarchy from the root group of the file [Gro15a] (see Figure 2.3).

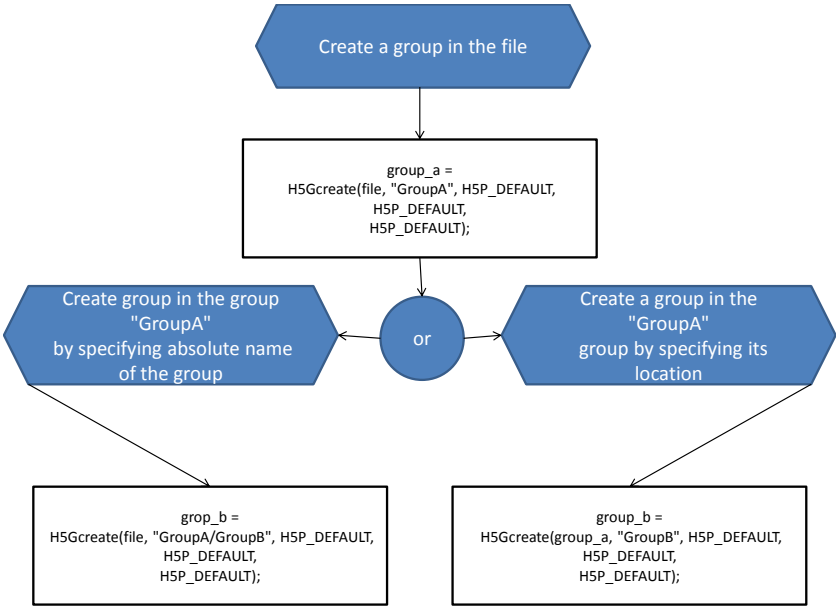

Figure 2.3: Creating an HDF5 group

The properties optionally specify settings for the group (or `H5P_DEFAULT` is used).

When a group is created or opened, following operations can be performed on the group [Groc]:

- Create new objects.
- Insert existing objects.
- Delete existing objects.
- Open and close member objects.
- Access information regarding member objects.
- Iterate across group members.
- Manipulate links.

The group is then closed.

2.2.4 Working with an HDF5 Dataset and with an HDF5 Attribute

Working with attributes is similar to working with datasets; therefore, the steps that are required to create and open a dataset can be applied for attributes.

Creating a dataset requires the following steps [Gro15a]:

1. Define a dataspace for a dataset.
2. Define a datatype for a dataset.

3. Obtain the location identifier.
4. Create and initialize the dataset.
    a) Optional: Write the dataset data.
    b) Optional: Close the datatype, dataspace, property list.
5. Close the dataset.

The following steps are required to open and read or write an existing dataset [Gro15a]:

1. Obtain the location identifier.
2. Obtain the dataset name or index.
3. Open the dataset.
    a) Optional: Get the dataspace and the datatype.
    b) Optional: Specify the memory type.
    c) Optional: Read and/or write data.
4. Close the dataset.

2.3 The Database VOL-Plugin for HDF5

The VOL allows writing plugins for accessing HDF5 objects. Internally, a plugin can implement the different storage mechanisms of objects in any way desired [MC14]. The main goal is that each plugin provides applications with the same HDF5 API and data access model.

2.3.1 VOL Structure

Each VOL must contain a structure of type `H5VL_class_t` . This structure is defined as follows (see Listing 2.3) [Cha14b]:

```
1  typedef struct H5VL_class_t {
2     H5VL_class_value_t value;
3     const char *name;
4     herr_t (*initialize)(hid_t vipl_id);
5     herr_t (*terminate)(hid_t vtpl_id);
6     size_t fapl_size;
7     void * (*fapl_copy)(const void *info);
8     herr_t (*fapl_free)(void *info);
9     /* Data Model */
10    H5VL_attr_class_t attr_cls;
11    H5VL_dataset_class_t dataset_cls;
12    H5VL_datatype_class_t datatype_cls;
13    H5VL_file_class_t file_cls;
14    H5VL_group_class_t group_cls;
15    H5VL_link_class_t link_cls;
16    H5VL_object_class_t object_cls;
17    /* Services */
18    H5VL_async_class_t async_cls;
19    herr_t (*optional)(void *obj, hid_t dxpl_id, void **req,
         ↪ va_list arguments);
20 } H5VL_class_t;
```

Listing 2.3: VOL class

The fields in lines 10 to 16 define classes that provide functionality for accessing HDF5 objects (file, attribute, dataset, group, link, object and named datatype). Other object types (dataspace, property lists) are memory space objects that are not stored or accessed through an HDF5 file [Cha14a].

HDF5 library functions map to the callbacks in the classes that the plugin needs to implement (see Table 2.5).

Prefix of the HDF5 API function	VOL Class
H5A	`H5VL_attr_class_t`
H5D	`H5VL_dataset_class_t`
H5F	`H5VL_file_class_t`
H5G	`H5VL_group_class_t`
H5L	`H5VL_link_class_t`
H5O	`H5VL_object_class_t`
H5T	`H5VL_datatype_class_t`

Table 2.5: Mapping the HDF5 API to the Callbacks

For example, the functions containing the prefix H5G map to one of the group callback routines in the `H5VL_group_class_t` class [Cha14b](see Listing 2.4).

```
1 typedef struct H5VL_group_class_t {
2    void *(*create)(void *obj, H5VL_loc_params_t loc_params,
        ↪ const char *name, hid_t gcpl_id, hid_t gapl_id, hid_t
        ↪ dxpl_id, void **req);
3    void *(*open)(void *obj, H5VL_loc_params_t loc_params, const
        ↪ char *name, hid_t gapl_id, hid_t dxpl_id, void **req);
4    herr_t (*get)(void *obj, H5VL_group_get_t get_type, hid_t
        ↪ dxpl_id, void **req, va_list arguments);
5    herr_t (*specific)(void *obj, H5VL_group_specific_t
        ↪ specific_type, hid_t dxpl_id, void **req, va_list
        ↪ arguments);
6    herr_t (*optional)(void *obj, hid_t dxpl_id, void **req,
        ↪ va_list arguments);
7    herr_t (*close) (void *grp, hid_t dxpl_id, void **req);
8  } H5VL_group_class_t;
```

Listing 2.4: Group class

For more information on the rest of the fields refer to the VOL user guide.

2.3.2 Mapping the API to the Callbacks

The callback interface defined for the VOL has to include all HDF5 API functions that potentially would modify the HDF5 file; however the API is extensive (containing hundreds of functions). That is why it was decided to not have a one-to-one mapping from the API to VOL callbacks [MC14].

Generic Callbacks

To provide applications with the same HDF5 API (by using one-to-N mapping), the VOL class implements generic callbacks. There are three generic callbacks for each model object [Cha14b]: Get-callbacks, Specific-callbacks, Optional callbacks.

Generic operations handle the three sets of the HDF5 library functions[Cha14b]:

- Get-operations that return certain information about an object will map to the get-callbacks.
- Specific-callbacks handle functions specific to each HDF5 object. Having a callback for each one would clutter the VOL callback structure.
- Optional callbacks handle new HDF5 functionality that is added specifically for certain plugins and it is not general enough to be implemented by most plugins.

Generic callbacks use an argument of type `va_list` to handle the different set of parameters that could be passed in. The get- and specific-callbacks also have an

argument that contain an enum of the type of a desired function. Using that type, the `va_list` argument can be parsed [Cha14b]. For example, Listing 2.5 demonstrates the get callback in the group class that should retrieve information about the group as specified in the `get_type` parameter [Cha14b].

```
1 herr_t H5VLgroup_get(void *obj, hid_t plugin_id,
     ↪ H5VL_group_get_t
2 get_type, hid_t dxpl_id, void **req, va_list arguments);
```

Listing 2.5: Group get function

The type `H5VL_group_get_t` is an enum defined as follows [Cha14b]:

```
1 typedef enum H5VL_group_get_t{
2     H5VL_GROUP_GET_GCPL,
3     H5VL_GROUP_GET_INFO
4 } H5VL_group_get_t;
```

Listing 2.6: Enum group get

The group-enum have two modes:

- `H5VL_GROUP_GET_GCPL` retrieves the group creation property list.
- `H5VL_GROUP_GET_INFO` retrieves the group information.

The argument of type `va_list` contains a variable list of arguments depending on `get_type` parameter and output pointers for the get-operation.

Create and Open Callbacks

All data model classes define a similar create and open callbacks:

- Create callbacks implement an object(group, dataset, attribute etc.) in the container of the location object and return an identifier of type `hid_t` for that object containing information to access the object in future calls [Cha14b]. In a case of the file callbacks, the location object is not needed.
- Open callbacks should open an object in the container of the location object and return a pointer to the object structure containing information to access the object in future calls [Cha14b].

The HDF5 library implements several functions to open and create object. For example, to create an attribute the functions `H5Aopen_by_name` or `H5Aopen_by_idx` could be used [Groa]. Such functions map to one open or create callback with a location parameter that indicates the access type. The location parameter is a structure that holds parameters for object locations.

Close Callbacks

Close callbacks should terminate access to an object and free all resources it was consuming, and return an `herr\t` indicating success or failure [Cha14b].

Summary

This chapter has given a necessary for the thesis description of the HDF5 abstract data model and the HDF5 programming model. The abstract data model defines HDF5 objects that have to be implemented. A programming model describes the relationships between HDF5 objects. These principles should be applied to design of the Database VOL Plugin.

This chapter has also introduced mapping the HDF5 API calls and the callback interface that has to be implemented in the Database VOL Plugin.

3 Related Work

In this chapter several works that use a related technique (separating data and metadata) will be presented.

3.1 Separation of metadata and data for other file formats

The method of a separating data and metadata are already used for other file formats. One example of these works is presented below.

The authors of [MI] demonstrate the MSD format as a potential extension to the DICOM format[1]. A DICOM data objects consist of image pixel data and associated metadata. This format doesn't allow separate metadata access; in cases that only need access to metadata, the DICOM format increases the running time (tag morphing is an example of one such use cases).

MSD stores information using two files rather than in many single frame files. MSD separates metadata from pixel data, and at the same time eliminates the replicated data elements. The first file contains the de-duplicated metadata, and the second contains pixel data. The results show that MDS significantly improves processing time for tag morphing.

3.2 Storage of HDF5 Objects in separate locations

In[KM12] VOL Plugin is presented, that stores every HDF5 objects in a separate location. Files and groups are stored as directories, whereas datasets and attributes are stored in files created using PLFS.[2]

For example, file `"test.h5"` contains group A, which itself contains dataset B. B contains attribute C. Thus these file and group are stored as following directories: `/test.h5/` and `/test.h5/A`. Dataset and attribute, which contain raw data, are stored in files at the following paths: `/test.h5/A/B` and `/test.h5/A/B.C` . The name of the attribute is stored as `parent object.attribute name` to denote the object to which the attribute is attached.

Results demonstrate that the plugin shows good performance.

[1]DICOM: Digital Imaging and Communications in Medicine format. This format is used for handling, storing, printing, and transmitting information in medical imaging.
MSD: Multi-Series DICOM

[2]Parallel Log Structured File System, which transforms N-1 access pattern into N-N

Summary

The presented works have shown that separating data and metadata improves I/O performance. Unlike the presented in [KM12] VOL Plugin, the Database VOL Plugin uses a database for metadata; thereby, it allows efficient storage of metadata using SQL language. Moreover, in case of the large number of metadata the implemented by authors of [KM12] plugin can overload the metadata server.

4 Design

In this chapter, the design of the separate storage of HDF5 data and metadata will be introduced. The abstract data model of HDF5 will be presented in a relational database.

4.1 Model of the Database VOL Plugin

HDF5 is suitable for efficient management of large and complex data. It supports complex relationships between objects. A parallel HDF5 application has multiple processes accessing a single file. Unfortunately, a large number of small and non-contiguous disk accesses typically results in a poor performance.

To improve the performance of the read and write operations, an approach is to store metadata in a database and data in a file system; then to access metadata separately on a faster storage device.

The VOL allows third party plugin development that can provide the efficient storage mechanisms of HDF5 objects. Figure 4.1 illustrates how HDF5 data are stored using the Database VOL Plugin.

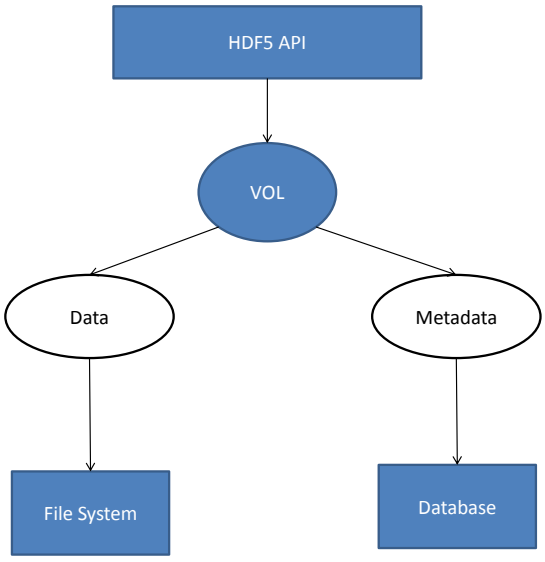

Figure 4.1: Using the Database VOL Plugin

Instead of storing all objects in a single file, the plugin stores data that are contained in datasets in a file system and metadata are stored in a relational database.

4.2 Storage of HDF5 Metadata in a Relational Database

Metadata represent the HDF5 objects and the relationships between them; therefore, the relational database is responsible for a maintenance of the HDF5 abstract data model.

Some relevant information about the relational database is presented below:

- A relational database is a set of tables of columns and rows containing data organized into categories.
- A table is a collection of objects of the same type (rows) with supporting metadata (columns).
- A primary key is a column in a table which contain unique values for each row in a database table; it makes it possible to select, add, delete or modify one and only one row in a table; therefore, most implementations have a unique primary key for each table.

4.2.1 Representation of HDF5 Objects

The relational database contains a set of tables that organize objects defined with the HDF5 data model. The goal of this thesis is to implement such VOL operations that handle the most common needs of HDF5. Therefore, some callbacks for the more complex aspects of accessing files are not designed and not implemented. For the same reason some object attributes are not presented in the tables. There are some common features:

- Objects in HDF5 have an unique identity within the hierarchy of the file and can be accessed by giving their path names. Therefore, each table can use the path name as a unique primary key.
- Each table has columns containing the information about primary data object[1].

Model of HDF5 objects with specific to each of them metadata are presented in detail below.

File

In the HDF5 data model the container for all objects is represented by a file; therefore, the HDF5 file has no primary data object and its name an initial point of each object path name.

Figure 4.2 represents a file model. The second column contains the information about the type of each attribute.

[1] direct parent object

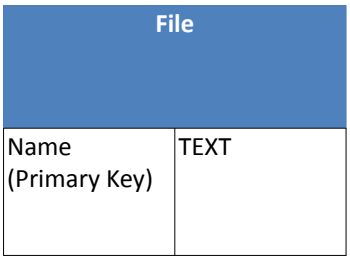

Figure 4.2: File

Group

A group is an association between HDF5 objects; a group's direct parent object can be a file (a root group) or other group.

Therefore, to implement groups the following attributes are needed:

- The name.
- The path name (a primary key).
- The information about primary data object: name and type.

Figure 4.3 represents a group with its attributes.

Group	
Path (Primary Key)	TEXT
Primary Data Object Type	TEXT
Primary Data Object Name	TEXT
Name	TEXT

Figure 4.3: Group

Dataset

A dataset is characterized by a dataspace and a datatype. Therefore, besides the dataset name, the path name (a primary key) and the primary data object type and name, the dataset needs attributes containing the information about dataspace and datatype. Knowing the number of dimensions and the type of data elements, a datasets size can be calculated (a `size` attribute). The HDF5 data are stored in a file system, so there is no table for them.

Figure 4.4 represents a dataset with its attributes.

Dataset	
Path (Primary Key)	TEXT
Primary Data Object Name	TEXT
Primary Data Object Type	TEXT
Name	TEXT
Type	BLOB
Space	BLOB
Data Size	INTEGER

Figure 4.4: Dataset

Attribute

An attribute consists of a name and data and looks very similar to datasets.

Attributes are assumed to be small, so storing them as datasets would be quite inefficient. Therefore, data are stored in a database and not in a file system (see Figure 4.5).

Attribute	
Path (Primary Key)	TEXT
Primary Data Object Name	TEXT
Primary Data Object Type	TEXT
Name	TEXT
Type	BLOB
Space	BLOB
Data Size	INTEGER

Attribute Data	
Path (Foreign Key)	TEXT
Data	BLOB

Figure 4.5: Attribute

4.2.2 Working with the Database VOL Plugin

Below is presented a programming model with the use of the Database VOL Plugin.

Create Object

Before HDF5 objects can be used, they must be created or opened. For example, when the file is created, the plugin initializes a row for this file in a File table (see Figure 4.6).

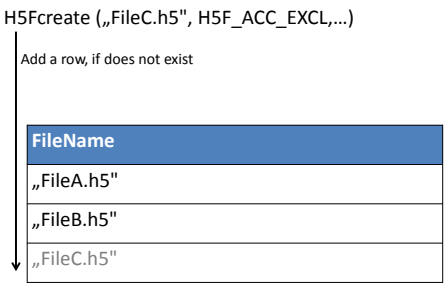

Figure 4.6: Open and Create Operations

Location Identifier

The HDF5 objects are implemented as data structures. When the object is created or opened, the plugin returns a special pointer - identifier; an identifier is used as a reference to a specific instance of an object (see chapter 2.2.1). That pointer can be used as an location identifier for the direct successors of the parent object; this provides a possibility to obtain information about primary data object and to create the path name.

Primary data object information is stored for each object in a database. In the example below the plugin obtains the information related to parent object type and name from object identifies `file` and `group` (see Figure 4.7).

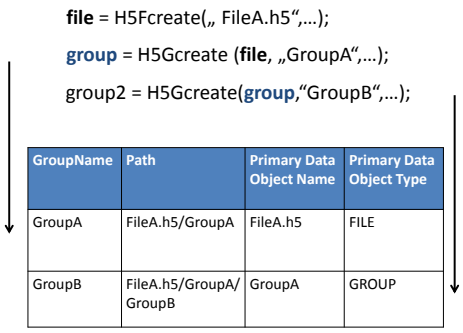

Figure 4.7: Primary Data Object Information

Dataset Object Routines

Some part of the object information is stored in data structures. The main VOL operations are explained below by the example of datasets; a special focus lies on a storing the dataset information (in a data structure and in a database).

When the dataset is created, the plugin initializes all attributes of Dataset table and the name, the location and the data size of the dataset structure (see Figure 4.8).

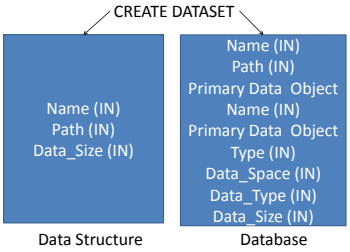

Figure 4.8: Create Operation

Open function creates path name and selects from a database the data size. Then the selected value and the name are stored in the dataset structure (see Figure 4.9).

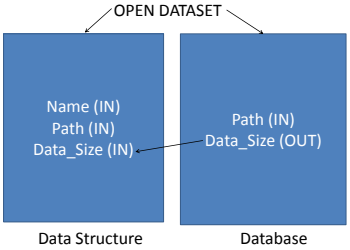

Figure 4.9: Open Operation

Get functions can return certain information about an object. In this case the plugin selects from a database the data type and the data space with the use of path-parameter from the data structure (see Figure 4.10).

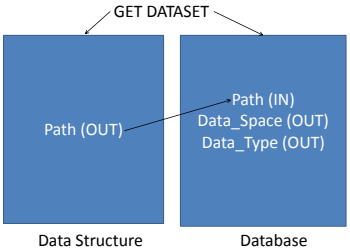

Figure 4.10: Get Operation

The Dataset table is not needed for write and read operations. The plugin use the information about the data size and the position of the data in the destination file from the dataset structure (see Figure 4.11).

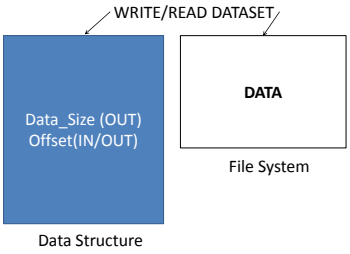

Figure 4.11: Write/Read Operations

Summary

In order to improve the performance of the HDF5 access operations, the Database VOL Plugin is responsible for a separation data that are contained in datasets and for managing HDF5 metadata in a database. The relational database implements the HDF5 abstract data model and makes it possible to manipulate with objects. To read and write data correctly in a destination file in a file system, the plugin uses data size and offset identifiers.

5 Implementation

In this chapter, an overview of the implementation of the Database VOL Plugin will be given. Firstly, the main plugin components will be introduced, followed by operations that are of particular interest:

- *Connection to a database and a data file.*
- *The dataset routines that allow to create and manage datasets in a data file.*
- *The storage of the attribute data in a database.*
- *The barrier synchronization of multiple access to data.*

5.1 The Main Components

In order to implement the desired storage of data and metadata as described in chapter 3, the plugin has followed main components:

- Calback operations. The plugin implements the main callbacks defined in the VOL class (see section 2.3.1). They capture the HDF5 API operations that would access the file. The callback operations call supporting database operations for managing metadata.
- Database operations. They allow accessing the database using the SQL query language. These functions are implemented with the use of SQLite. SQLite provides a disk-based database that does not require a separate server process. It is possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL.
- Data structures. There are data structures for file, group, dataset and attribute objects, which are used by callbacks operations as a reference to a specific instance of each object.
- Fapl operations. They are defined in the VOL class and used to store the info data that the plugin needs.

5.2 Create/Open/Close the Database and the Data File

It is reasonable to create (or open) and close the database and the data file only one time when the file create/open/close operations are called.

An example of a database and data file create routine is shown in Listing 5.1. The function `H5Pget_vol_info` (line 5) returns the info data, that contains the information received from an application about a database name and a data file name (`ginfo->db_fn`, `ginfo->data_fn`).

A database connection is opened with the `sqlite3_open` function (line 8). Then the database operation `file_create_database` is called (line 12). This function creates tables for all objects and insert values in a File table.

The function `open64` creates a file description that refers to a data file (line 15).

The plugin stores an SQLite database handle and a descriptor for a data file in the `SQF_t` structure. The struct also contains complex data type `SQF_t` in variable `root` (line 17), it gives a possibility for other routines to access a connection to the database and the data file.

```
1  static void *
2  H5VL_extlog_file_create(const char *fname,...)
3  {
4      ...
5      ginfo = (h5sqlite_fapl_t*)H5Pget_vol_info(fapl_id);
6      ...
7      SQF_t *file;
8      int rc = sqlite3_open(ginfo->db_fn, &file->db);
9      ...
10     if (0 == ginfo->mpi_rank)
11     {
12     file_create_database(fname, flags, file->db);
13     }
14     ...
15     file->fd = open64(ginfo->data_fn, O_RDWR |O_CREAT, 0666 );
16     ...
17     file->root = file
18     ...
19 }
```

Listing 5.1: Create the database connection and the data file

5.3 Dataset Create Routine

The plugin needs datatype, dataspace and data size for writing and reading datasets (see section 3.2.1).

Listing 5.2 demonstrates how these values can be created. The data size argument is calculated using the number of dimensions in a dataspace, the size of each dimension

and the size of elements (lines 16-17).

The argument `dcpl_id` contains the dataset datatype (an `hid_t`) and dataspace (an `hid_t`) identifiers. The function `H5Pget` retrieves a copy of the value for a property dataspace and datatype in a property list (lines 6, 13). The number of dimensions in a dataspace is determined with `H5Sget_simple_extent_ndims` function; `H5Sget_simple_extent_dims` returns the size and maximum sizes of each dimension (lines 7, 10). `H5Tget_size` returns the size of a datatype in bytes (line 14).

```
1  static void *
2  H5VL_extlog_dataset_create(..., hid_t dcpl_id, ...)
3  {
4      ...
5      hid_t space_id;
6      H5Pget(dcpl_id, "dataset_space_id", &space_id);
7      int ndims = H5Sget_simple_extent_ndims(space_id);
8      hsize_t maxdims[ndims];
9      hsize_t dims[ndims];
10     H5Sget_simple_extent_dims(/* in */ space_id, /* out */
          ↪ dims, /* out */maxdims);
11
12     hid_t type_id;
13     H5Pget(dcpl_id, "dataset_type_id", &type_id);
14     size_t type_size = H5Tget_size(type_id);
15     size_t data_size = type_size;
16     for(int i = 0; i < ndims; ++i)
17     {
18         data_size *= dims[i];
19     }
20     ...
21 }
```

Listing 5.2: Creation arguments for attribute data

The plugin stores some part of the object information in data structures. The dataset structure is defined as follows (see Listing 5.3):

```
1  typedef struct SQD_t
2  {
3      SQF_t* root;
4      char *name;
5      char *location;
6      int position;
7      size_t data_size;
8  } SQD_t;
```

Listing 5.3: Structure for dataset

Listing 5.4 demonstrates the initialization of the dataset structure in the create dataset operation. The function receives the name as input parameter; the data size is calculated

as described before. With the use of the `switch` statement the plugin gets the type of a primary object. Then, the `root` and the path variables are initialized (lines 14, 16). The last one is created using an object name and a parent name with `snprintf` function (lines 15-16).

```
1  static void *
2  H5VL_extlog_dataset_create(void *obj,.., const char *name, ...)
3  {
4      ...
5      SQD_t *dset;
6      ...
7      dset->name = strdup(name)
8      dset->data_size = data_size;
9      switch(loc_params.obj_type)
10     {
11         case H5I_FILE:
12         {
13         SQF_t *o = (SQF_t *)obj;
14         dset->root = o->root;
15         snprintf(ch, sizeof ch, "%s%s", o->name,"/");
16         snprintf(dset->location, 512, "%s%s", ch, name);
17         dataset_create_database(name, o->name, "H5I_FILE",
               ↪ dset->location, type_id, space_id,data_size,
               ↪ dset->root->db);
18     }
19     ...
20     return (void *)dset;
21 }
```

Listing 5.4: Initialization of values in a dataset structure

Dataset create function calls a supporting database operation, that insert values in Dataset table (see Listing 5.4 , line 17). How the plugin stores the data type and data space values in a table is of particular interest for this thesis and is shown in Listing 5.5.

Firstly, it is important to make an overview of the `sqlite3` module functions. `sqlite3_prepare` compiles an SQL statement into a byte-code (line 6). `sqlite3_bind` functions bind values to place-holders (lines 12, 22, 23). The prepared statement is executed with `sqlite3_step` (line 25). Finally, `sqlite3_finalize` destroys the prepared statement object (line 26).

To make it possible for next access sessions to reconstruct a data type description (`type_id`) and a data space description (`space_id`) the functions `H5Tencode` / `H5Sencode` and `H5Tdecode` / `H5Sdecode` are used. `H5Tencode` and `H5Sencode` convert a data type description and a data space description into binary form in a buffer (lines 11, 21). Using this binary form in the buffer, these objects can be reconstructed using `H5Tdecode` to return a new object handle (`hid_t`).

```
1  void dataset_create_database(...,hid_t type_id, hid_t space_id,
       ↪ hid_t data_size,...)
2  {
3      ...
4      char *sql2=  "INSERT INTO DATASET VALUES(?,?,?,?,?,?,?);";
5      ...
6      rc = sqlite3_prepare(db, sql2, strlen(sql2), &res, &pzTest);
7      ...
8      size_t type_size;
9      H5Tencode(type_id, NULL, &type_size);
10     char type_buf[type_size];
11     H5Tencode(type_id, type_buf, &type_size);
12     sqlite3_bind_blob(res, 5, type_buf, type_size, NULL);
13     ...
14     int rank = H5Sget_simple_extent_ndims(space_id);
15     hsize_t dims[rank];
16     hsize_t max_dims[rank];
17     H5Sget_simple_extent_dims(space_id, dims, max_dims);
18     size_t space_size;
19     H5Sencode(space_id, NULL, &space_size);
20     unsigned char* space_buf = (unsigned char*)
         ↪ malloc(space_size);
21     H5Sencode(space_id, space_buf, &space_size);
22     sqlite3_bind_blob(res, 6, space_buf, space_size, NULL);
23     sqlite3_bind_int64(res, 7,data_size);
24     rc = sqlite3_step(res);
25     sqlite3_finalize(res);
26 }
```

Listing 5.5: Part of the Insert operation for the Dataset table

5.4 Dataset Open Routine

Dataset open callback initializes name, root and location `struct` values in the same way as create callback; the data size is retrieved from the database.

5.5 Dataset Get Routine

Listing 5.6 shows some part of the dataset get routine implementation.

The get callback returns information about the dataset as specified in the `get_type` parameter. The `dataset_get_database` function selects data type and data space from the table, reconstructs their binary form and returns new data descriptions (line 5).

Then, the switch statement selects the type; `va_list arguments` contains a variable list depending on that type. `va_arg` macro expands to an expression of type `hid_t` with

the value of the current argument in the argument list (line 11). Finally, that buffer for the identifier is initialized with a required information (line 12).

```
1  static herr_t
2  H5VL_extlog_dataset_get(void *dset, H5VL_dataset_get_t
       ↪ get_type,..., va_list arguments)
3  {
4      hid_t  *ret_id;
5      dataset_get_database(..., &type_id, &space_id, ...);
6      switch(get_type)
7      {
8          ...
9          case H5VL_DATASET_GET_SPACE:
10         {
11         ret_id = va_arg (arguments, hid_t *);
12         *ret_id = space_id;
13         }
14         break;
15         ...
16     }
17     ...
18 }
```

Listing 5.6: Dataset get callback

5.6 Dataset Read/Write Routine

The plugin supports parallel writing/reading datasets.

Listing 5.7 demonstrates some important parts of the dataset write function; the main focus is on an ability to write data in parallel. Firstly, the data size is calculated in the same way as in the dataset create routine (lines 7-16).

Then the current file position to write a portion of a dataset is found using values `mpi_rank` and `offset` from info structure (lines 17-19). The type `off64_t` allows to manipulate the file position of files that are larger than 2 gigabytes (line 19). The function `pwrite64` writes the specified number of bytes from the buffer to the file descriptor into the specified position.

In the while loop the number of bytes `count` and the specified position (`curr_offset` + `bytes_written_total`) are calculated for each parallel writing process and passed to the function (lines 26-31). The file descriptor is contained in the variable `root` of the dataset structure (`d->root->fd`). The loop is repeated until all the bytes from the buffer to the file are written.

Finally, the plugin updates a dataset file position; this value can be used for other dataset write operations (lines 34-35).

```
1  static herr_t
2  H5VL_extlog_dataset_write(void *dset, hid_t mem_type_id, hid_t
       ↪ mem_space_id,..., const void *buf, ...)
3  {
4      ...
5      SQD_t *d = (SQD_t *) dset;
6
7      int rank = H5Sget_simple_extent_ndims(mem_space_id);
8      hsize_t dims[rank];
9      hsize_t max_dims[rank];
10     H5Sget_simple_extent_dims(mem_space_id, dims, max_dims);
11     size_t block_size = H5Tget_size(mem_type_id);
12     assert(block_size != 0);
13     for (size_t i = 0; i < rank; ++i)
14     {
15         block_size *= dims[i];
16     }
17     size_t rel_offset = block_size * ginfo->mpi_rank;
18     ...
19     off64_t curr_offset = ginfo->offset + rel_offset;
20     size_t bytes_written_total = 0;
21     size_t count = 0;
22     ssize_t bytes_written = 0;
23
24     while (bytes_written_total <= block_size)
25     {
26         size_t bytes_left = block_size - bytes_written_total;
27         count = (COUNT_MAX < bytes_left) ? COUNT_MAX :
               ↪ bytes_left;
28         ...
29         bytes_written = pwrite64(d->root->fd, buf +
               ↪ bytes_written_total, count, curr_offset +
               ↪ bytes_written_total);
30         ...
31         bytes_written_total += bytes_written;
32     }
33     ...
34     d->position = ginfo->offset;
35     ginfo->offset = ginfo->offset + d->data_size;
36     ...
37 }
```

Listing 5.7: Dataset write routine

5.7 Attribute Read/Write Routine

Listing 5.8 shows how the plugin reads data from the database. The SQL statement to select data using path name is demonstrated in line 4. Attribute data are stored in a database in a binary format (BLOB type). `sqlite3_column_blob()` forces the result into the desired format, then `sqlite3_column_bytes()` returns the number of bytes in that BLOB (lines 6-7). Finally, `memcpy` copies the values of `data_size` bytes from the location pointed to by `data` directly to the memory block pointed to by `buf`.

```
1  void attr_read_database( const char *location, /*OUT*/void
       ↪ *buf, sqlite3* db)
2  {
3      ...
4      char *sql = "SELECT data FROM DATA WHERE Path=?;";
5      ...
6      const void * data =  sqlite3_column_blob(res, 0);
7      int data_size = sqlite3_column_bytes(res, 0);
8      memcpy(buf, data, data_size);
9      ...
10 }
```

Listing 5.8: Attribute read database function

5.8 The barrier synchronization

Strict time limits for the bachelor thesis do not allow to implement all HDF5 objects: the actual implementation does not support the Link interface (`H5L`). The lack of this implementation can result in some access problems.

The problems and the implemented solutions to these problems are demonstrated below by the example of running the routine `H5Dcreate H5Dwrite H5Dclose H5Dopen H5Dclose H5Ldelete`.

Scenario 1. Let us assume, that the process 1 needs relatively more time to write data. It tries then to open deleted by process 0 dataset (see Figure 5.1). To avoid this problem `MPI_Barrier` function after SELECT statement in `H5Dopen` can be used (see Figure 5.2).

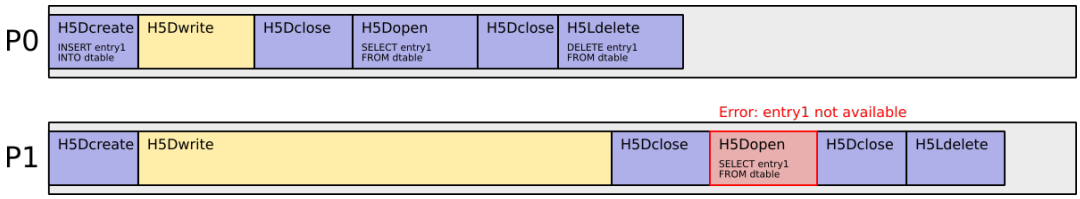

Figure 5.1: Error by open deleted file

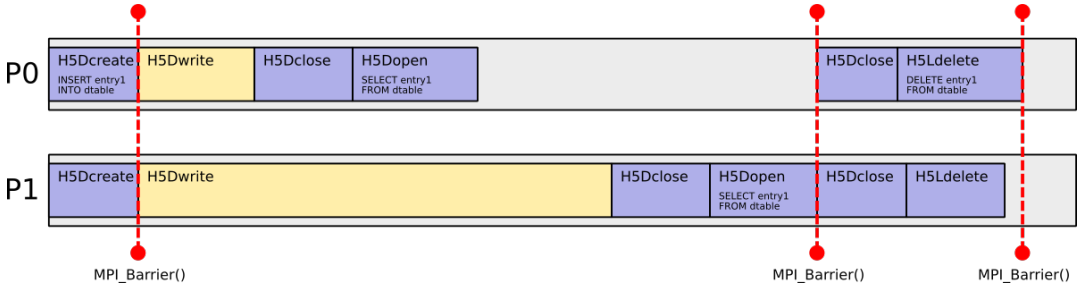

Figure 5.2: Error by open deleted file: Barrier solution

Scenario 2. Let us assume, that the process 0 for some reason needs more time for `H5Dcreate` operation. It is possible, that the process 1 tries to select some value using `H5Dopen`, but this value does not exist yet (see Figure 5.3). `MPI_Barrier` function after INSERT statement in `H5Dcreate` is a solution for this case (see Figure 5.4).

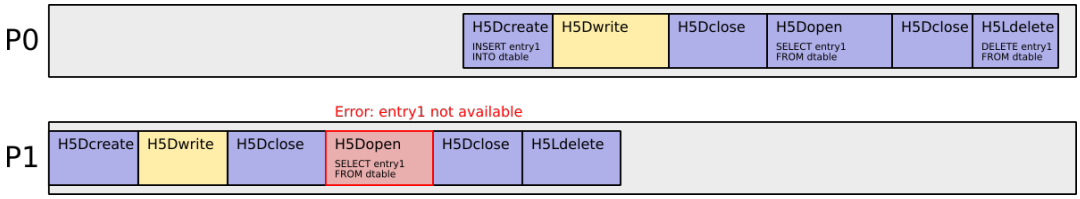

Figure 5.3: Error by open values that not exists yet

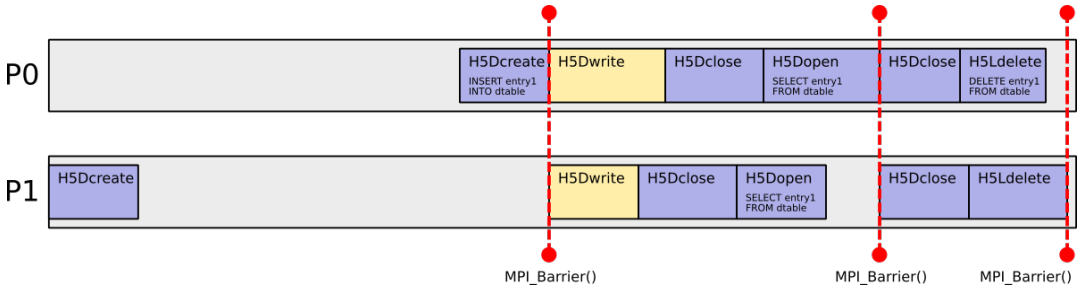

Figure 5.4: Error by open values that not exists yet: Barrier solution

There are a lot of optimization solutions for this implementation. For example, it is reasonable to use problem `MPI_Barrier` function after SELECT statement only for cases with the followed operation, that can modify a database. For read-only modus `MPI_Barrier` is not needed at all, since all read operations are idempotent.

In current implementation these optimization solutions do not used, because they do not relevant for the main goal of the thesis.

Summary

This chapter has shown that the Database VOL Plugin implements all important callbacks for HDF5 library. It separates metadata and data, stores data in a file system, and manages HDF5 metadata in a database using SQLite. The plugin also supports parallel HDF5 access using MPI library.

6 Evaluation

In this chapter the benchmark test and the obtained results will be introduced. Firstly, the test environment will be presented, followed by test configurations (the libraries, the data, the measured operations). Finally, the test will be run using original Native Plugin and Database VOL Plugin. The performance of these experiments will be compared and evaluated.

6.1 Test Environment

All experiments are conducted on the cluster of the Scientific Computing research group at the University of Hamburg (see Table 6.1).

CPU	2x Intel Xeon Westmere X5650
RAM	12GB (DDR3/PC1333)
Kernel	Linux cluster 4.4
NIC	2x Intel 82574L gigabit (Gbit) Ethernet
FS	Lustre 2.9.0

Table 6.1: Experiment environment

The benchmark is performed with 1, 8, 16, 24 processes per node. All experiments are run using OpenMPI 1.10.2 library. SQLite is compiled using `-DSQITE\_THREADSAFE=2` parameter to select serialized mode. In this mode SQLite can be safely used by multiple threads with no restriction.

6.2 Test Configurations

Parallel I/O

The HDF5 library can provide parallel support using the MPI library. A file can be opened in parallel from an MPI application by specifying a parallel file driver with an MPI communicator and info structure. This information is communicated to HDF5 through a property list [Groe]. In Listing 6.1 a file access property list is created and set to use the MPI-IO file driver.

```
1 hid_t fapl_id = H5Pcreate(H5P_FILE_ACCESS);
2 H5Pset_fapl_mpio(fapl_id, comm, info);
3 file_id = H5Fcreate(FNAME, H5F_ACC_TRUNC, H5P_DEFAULT, fapl_id);
```

Listing 6.1: Create the file in parallel

HDF5 allows to read or write to a portion of a dataset by use of hyperslab selection by selecting a subset of the dataspace in the file, selecting a local memory dataspace, and then using the memory and file dataspaces to read from or write to the dataset [KA10]. Firstly, the `H5Dget_space` obtains the dataspace of a dataset in a file; subset of that dataspace can be selected with `H5Sselect_hyperslab` (see Listing 6.2).

```
1 filespace = H5Dget_space(dset_id);
2 H5Sselect_hyperslab(filespace, H5S_SELECT_SET, offset, NULL,
    ↪ count, NULL);
```

Listing 6.2: Select hyperslab

Then each process defines dataset in memory and writes it to the hyperslab in the file (see Listing 6.3).

```
1 count[0]  = dimsf[0] / mpi_size;
2 count[1]  = dimsf[1];
3 offset[0] = mpi_rank * count[0];
4 offset[1] = 0;
```

Listing 6.3: Writing dataset by rows

Before reading a subset from or writing a subset to a dataset, in addition to a file dataspace the local memory space must be created (see Listing 6.4).

```
1 memspace = H5Screate_simple(RANK, count, NULL);
```

Listing 6.4: Create memory space

The local memory and file dataspace identifiers from the selections and property list for collective dataset write/read are passed into the read or write operation (see Listing 6.5).

```
1 /* Create property list for collective dataset write */
2 plist_id = H5Pcreate(H5P_DATASET_XFER);
3 ...
4 status = H5Dread(dset_id, ..., memspace, filespace, ..., ...);
```

Listing 6.5: Collective dataset write

Read/Write Tests

The test contains an enum of the type of operations that are requested to be performed:

- `IOTYPE_WRITE` (`H5Fcreate`, `H5Gcreate`, `H5Dcreate`, `H5Dwrite`, `H5Acreate`, `H5Awrite`).

- `IOTYPE_READ (H5Fopen, H5Gopen, H5Dopen, H5Dread, H5Aopen, H5Aread)`.

For each of these operations the functions `time_start` and `time_stop` take the performance measurement using `time` library.

The test can be run using the Database VOL Plugin or original Native Plugin. Listing 6.6 demonstrates the routine that registers the database plugin. Firstly, the info data including database name and file name are passed (lines 2-4). The plugin is registered with the `H5VLregister name()` (line 5). The application then sets the plugin access property in the file access property list (line 7).

```
1 #if defined VOL
2 h5sqlite_fapl_t finfo;
3 finfo.db_fn = "metadata.db";
4 finfo.data_fn = "data.dat";
5 hid_t vol_id = H5VLregister_by_name ("extlog");
6 ...
7 H5Pset_vol(fapl_id, vol_id, &finfo);
8 #endif
```

Listing 6.6: Register the Database VOL Plugin

The Data used for Experiments

The experiments are conducted on one node with different numbers of processes (1, 8, 16, 24) and repeated 10 times. Each write experiment creates file, group, attribute and 2-dimensional dataset with the 107520 x 4993 dimension size. The total file size is 2000 MiB. In every run, the data is distributed equally among the processes.

Read experiments open objects and read dataset, that was created by write experiments.

6.3 Test Results and Evaluation

Some outliers are not shown in pictures, because they have large values and would affect the presentation of results. The important outliers are explicitly stated in the text.

6.3.1 Metadata performance

The test measures the time is needed for the execution of all main operations and also ensures the plugin functions correctness. The followed metadata operations are controlled: H5Fcreate, H5Gcreate, H5Dcreate, H5Acreate, H5Awrite (create operation), H5Fopen, H5Gopen, H5Dopen, H5Aopen, H5Aread (read operation).

Results show that all these operations are correctly run. It is also important to mention, that the plugin writes and reads right attribute data using a database.

The figures below compares the time for some read and write operations using the database plugin, labeled `SQLite3-Plugin`, and that for the native plugin (see Figures 6.1, 6.2, 6.3). It can be seen, that the database plugin consistently shows good performance,

moreover, for some operation it outperforms the native plugin. In particular `H5Fopen` is much faster, because the database plugin has almost no initialization overhead. Native plugin slower, probably, because it has to initialize some internal buffers and preload metadata. For `H5Fcreate` it can be seen the opposite. Database plugin gets slower because it has to create the database. The native plugin gets faster, probably there is not many data to preload.

Overall, in group operations the database plugin is faster, because in contrast to the native plugin it does not need to make many initializations.

In dataset read and write functions the database plugin is slower, since it has to make a lot of operations to initialize the database (`H5Dcreate`) and to search values in it (`H5Dopen`).

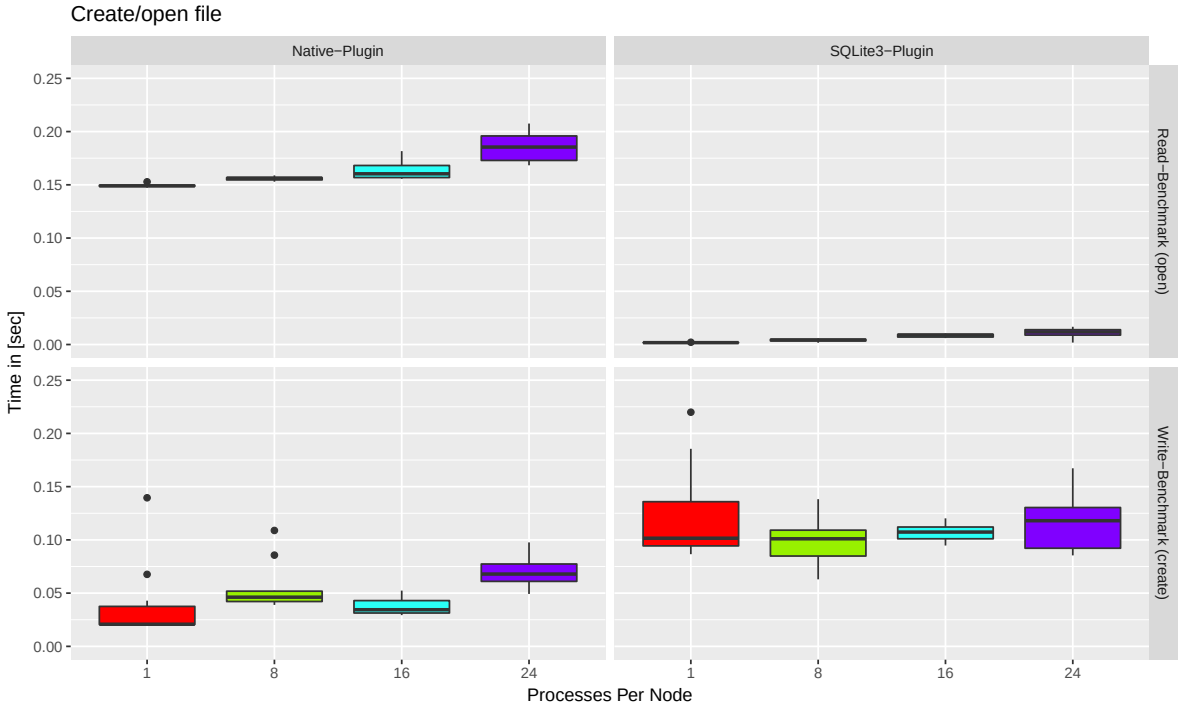

Figure 6.1: Create/open file

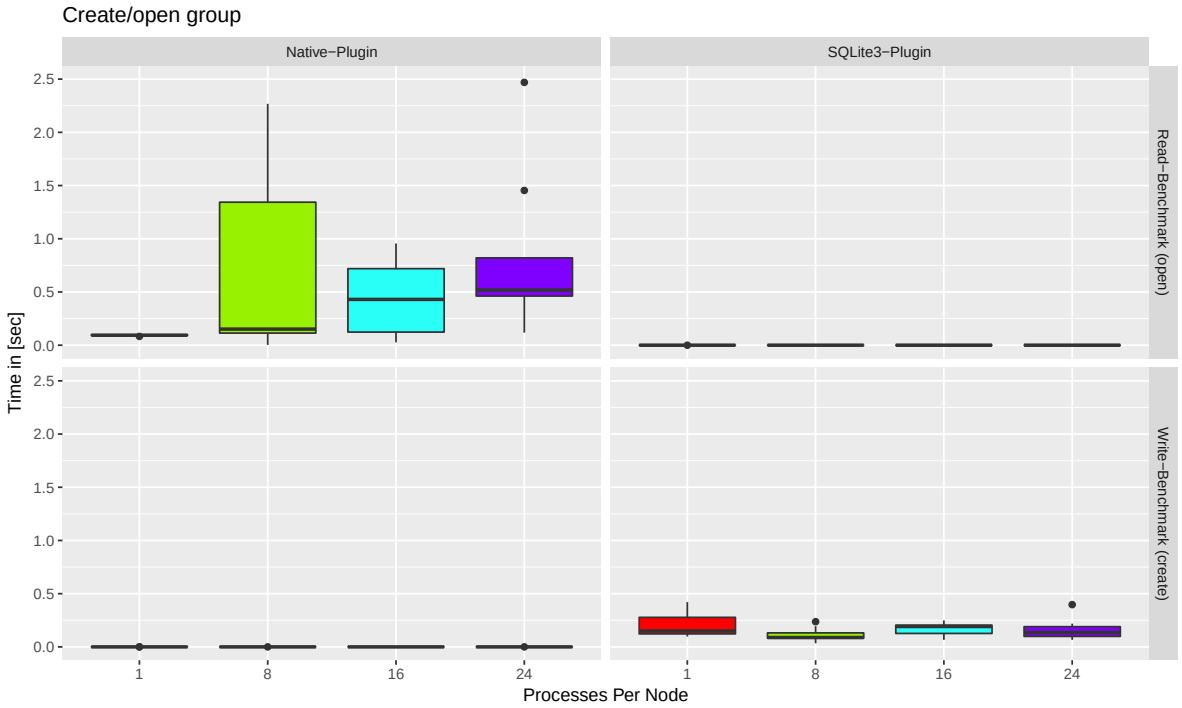

Figure 6.2: Create/open group

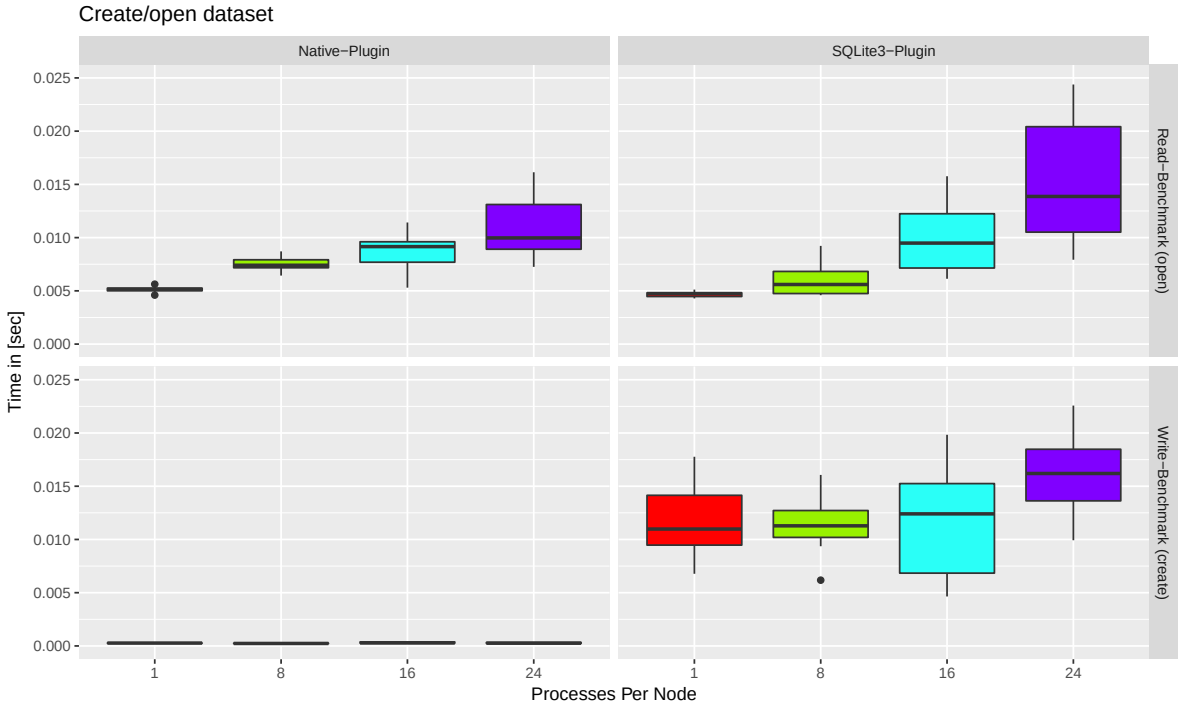

Figure 6.3: Create/open dataset

6.3.2 Data performance

Figure 6.4 compares the time for reading and writing the data in a file using the native and the database plugins.

Overall, these results show that the database plugin also for dataset write/read opeartions consistently shows good performance, however it does not outperform the native plugin. This is due to the fact that both plugins always saturate the network performance.

In experiments a small dataset (2GB) is compared to available amount of RAM (12GB). The reason is that on current system and with MPI implementation the maximum datasize that can be created by one process is limited to 2GB/dataset is used for experiments. That means, the dataset is probably not large enough to overwrite the internal cache of native plugin and some cache issues should be expected. Although, this problem can be solved by writing several datasets or by using more processes, but on this system the plugins will always saturate the network bandwidth. It is more interesting to observe the cache effects. Figure 6.4 does not show the outliers, for example, in case of PPN=24 it could even observed a write performance of 162 MB/s, which exceeds by far the network performance due the usage of internal cache. The maximum write performance achieved by SQLite3-plugin is 111 MB/s. This does not make the native plugin faster, because as can be seen in Figure 6.5 this data are written to the file, when the file is close, which results in large close times.

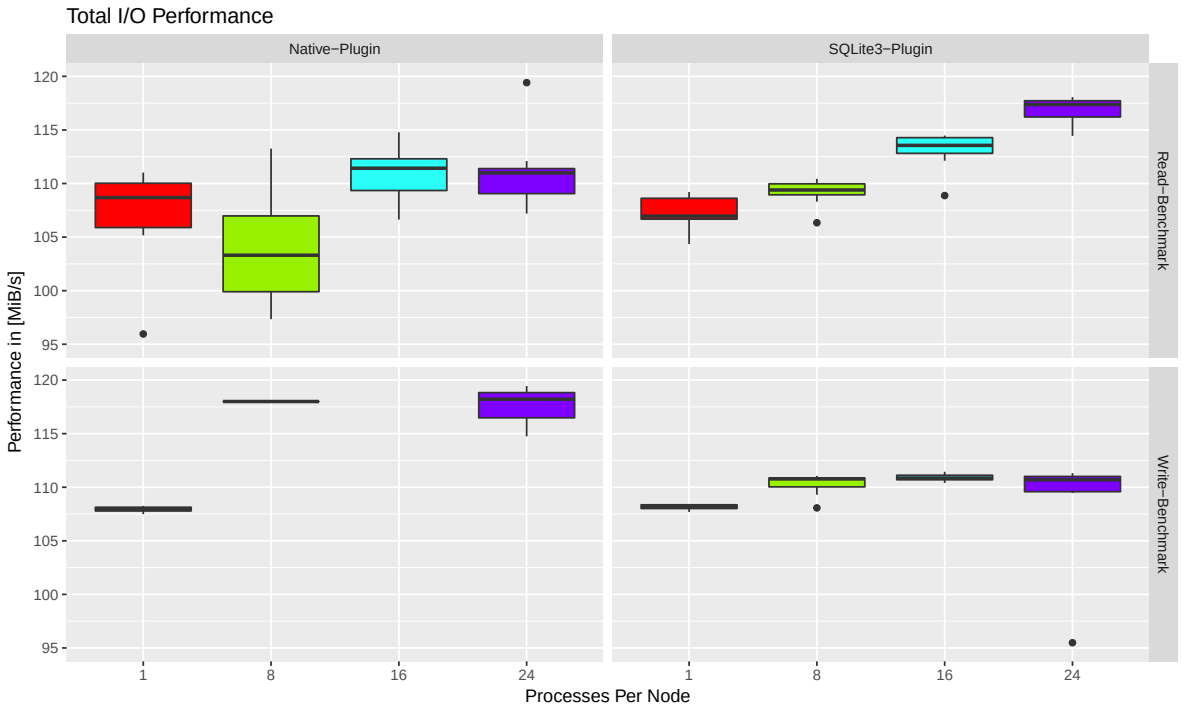

Figure 6.4: Total I/O Performance

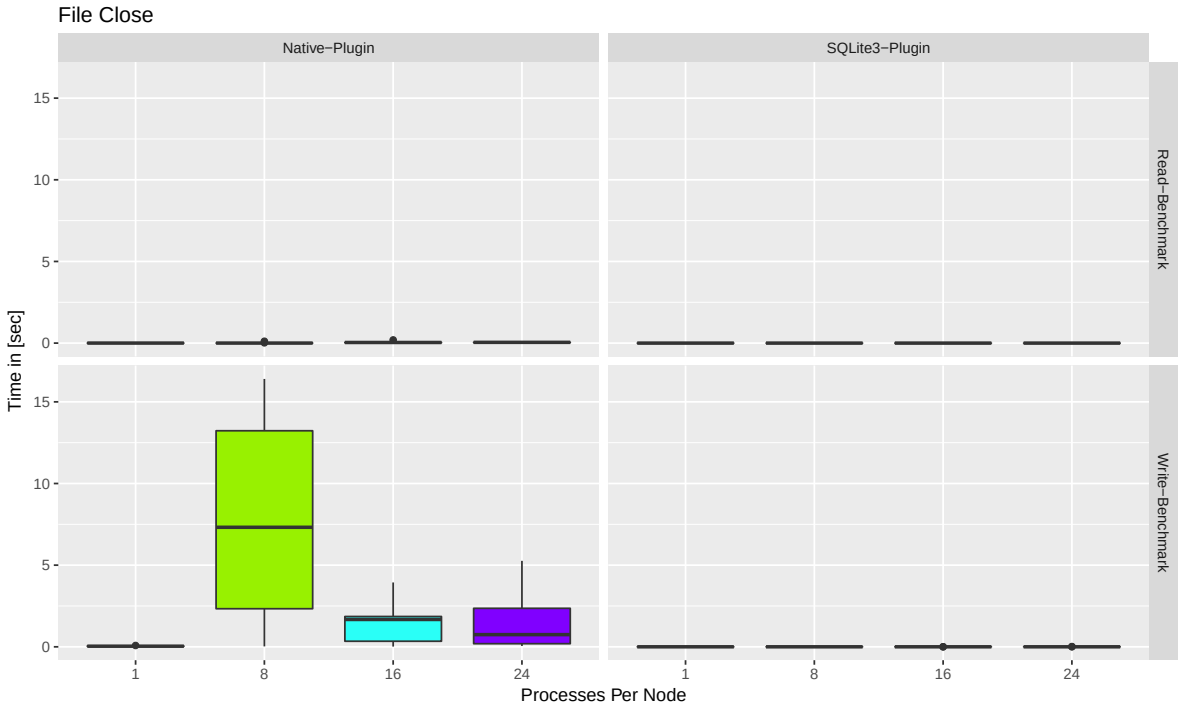

Figure 6.5: Close Performance

Besides, it can be also seen that compare to the native plugin, the database plugin has a low variance of the data read/write performance. This positive property makes the program execution easy to calculate.

6.3.3 Metadata: HDF5 vs. scientific applications

The output in Listing 6.7 was produced by the `h5dump` tool, which was used to dump the header of a HDF5 file. It shows how labels can be attached to a dataset.

```
1  HDF5 "grid.nc" {
2   GROUP "/" {
3       DATASET "lat" {
4           DATATYPE  H5T_IEEE_F32LE
5           DATASPACE  SIMPLE { ( 6 ) / ( 6 ) }
6           ...
7       }
8       DATASET "lon" {
9           DATATYPE  H5T_IEEE_F32LE
10          DATASPACE  SIMPLE { ( 5 ) / ( 5 ) }
11          ...
12      }
13      DATASET "time" {
14          DATATYPE  H5T_IEEE_F32LE
15          DATASPACE  SIMPLE { ( 4 ) / ( H5S_UNLIMITED ) }
```

```
16          ...
17      }
18      DATASET "var1" {
19          DATATYPE  H5T_STD_I32LE
20          DATASPACE  SIMPLE { ( 4, 6, 5 ) / ( H5S_UNLIMITED, 6,
                ↪ 5 ) }
21          ATTRIBUTE "DIMENSION_LIST" {
22              DATATYPE  H5T_VLEN { H5T_REFERENCE {
                    ↪ H5T_STD_REF_OBJECT }}
23              DATASPACE  SIMPLE { ( 3 ) / ( 3 ) }
24              DATA {
25                  (0): (DATASET 8428 /time ), (DATASET 10954
                        ↪ /lat ),
26                  (2): (DATASET 11377 /lon )
27              }
28          }
29          ...
30      }
31  }
32  }
```

Listing 6.7: Data and metadata [EB]

The dataset `var1` contains data. Its attribute `DIMENSION_LIST` includes three references (Links) to other datasets. `lon` and `lat` contain longitude and latitude values. Time values are stored in `time`. The meaning of these references is defined by the applications, for example in this case they are interpreted as labels to the dimensions.

It is disputable whether linked datasets belong to metadata or not. From the scientific application's point of view, axis labels are part of metadata, since they contain supporting information. For example, they can be used in search services to find required data (see use case 2 from section 1.4). For HDF5 these datasets are not part of metadata, since they are stored in ordinary datasets, especially because the large size does not allow storage in attribute object.

It would be not reasonable to store datasets in SQLite3 database in binary format, because of their size and how they are used by scientific applications it would probably result in a slower performance. As already mentioned in use case 1, each process requests only the parts of metadata, that are required for the processing of data. If a database stores all the metadata in binary format, then it can be accessed only as the whole object.

One possible solution to the problem could be the following. Datasets can be taken to pieces and stored element-wise in database, but such method is much more complicated and error-prone. A better way would be to store datatsets in binary format, just as it was produced by the HDF5 library. Probably, a much more better solution would be to store the data in some machine independent format.

The scientific application's point of view is more important for the goals of this thesis; therefore, it is doubtful whether the SQLite3 the optimal allocation for attributes.

Summary

In this chapter the results of the benchmark test were presented and discussed. The test results have shown, that the Database VOL - plugin correctly runs all important HDF5 operations. Write and read operations for attributes and datasets write/read the right data. The I/O performance using the database plugin was compared to the native plugin.

Overall time performance results are consistently good.

In this chapter was also presented a critical discussion of the approach, since metadata can be seen from different viewpoints (scientific applications vs. HDF5).

7 Conclusion and Future Work

This chapter summarizes and concludes the thesis. Additionally, an outlook regarding future work will be presented.

7.1 Conclusion

The main goal of this thesis is to implement the Database VOL-plugin for HFD5, that allows storing HDF5 data and metadata in different locations. The current implementation of the plugin provides applications with the main HDF5 API operations that would access the file. The plugin callback functions store HDF5 metadata in a database and data in a file system.

Outsourcing of metadata to a SQLite3 database implies some additional overhead. The main reasons are access to shared file system and `MPI_Barriers`. For the latter, there is a lot of optimization potential.

Scientific applications and HDF5 have a different view on metadata. This makes outsourcing of metadata to SQLite3 a kind of difficult. From point of view of scientific applications, in addition to HDF5 metadata, the database must contain some datasets, which causes some troubles, for axample for performance reasons, the datasets can not be stored in binary format, but must be stored by individual elements.

7.2 Future Work

Firstly, it would be reasonable to port the code to a database server (for example, PostgreSQL). It would significantly reduce the amount of file access and would simplify the implementation of a search service.

The actual implementation also does not support the Link interface (`H5L`). This interface includes functions that enable the creation and use of link classes in HDF5, and is required to create references.

Using such model the plugin can store in a database for each parts of a dataset supporting parts of metadata. It eliminates the problem of a large number of small and non-contiguous disk accesses and improves I/O performance.

As already mentioned, the separation of metadata and data is a first step to the high performance HDF5 plugin. The current plugin implementation allows the next step, the parallel I/O of dataset, which might be a quite challenging task.

Bibliography

[Cha14a] Mohamad Chaarawi. A Developer's Guide for the HDF5 Virtual Object Layer. 09 2014.

[Cha14b] Mohamad Chaarawi. User Guide for Developing a Virtual Object Layer Plugin. 09 2014.

[CT] NASA Curt Tilmes. Data Formats: Using self-describing data formats. Last accessed: 2017-05.

[EB] Dr. Julian Kunkel Eugen Betke. External Links for netCDF. Last accessed: 2017-07.

[Groa] The HDF Group. H5A: Group Interface. Last accessed: 2017-04.

[Grob] The HDF Group. H5F: File Interface. Last accessed: 2017-05.

[Groc] The HDF Group. H5G: Group Interface. Last accessed: 2017-04.

[Grod] The HDF Group. HDF5: API Specification Reference Manual. Last accessed: 2017-04.

[Groe] The HDF Group. Introduction to Scientific I/O. Last accessed: 2017-06.

[Grof] The HDF Group. The File Interface(H5F). Last accessed: 2017-05.

[Gro15a] The HDF Group. HDF5 User's Guide. 2015. Last accessed: 2017-05.

[Gro15b] The HDF Group. Virtual Object Layer in HDF5. 05 2015. Last accessed: 2017-05.

[Gro16] The HDF Group. High Level Introduction to HDF5. 09 2016. Last accessed: 2017-06.

[KA10] The HDF Group Katie Antypas. Intro to HDF5. 10 2010. Last accessed: 2017-04.

[KM12] Aaron Torres Gary Grider Edgar Gabriel Kshitij Mehta, John Bent. A Plugin for HDF5 using PLFS for Improved I/O Performance and Semantic Analysis. 1 2012. Last accessed: 2017-06.

[MC14] Quincey Koziol Mohamad Chaarawi. RFC: Virtual Object Layer. 09 2014.

[MI] James Philbin Mahmoud Ismail, Yu Ning. Separation of metadata and bulkdata to speed DICOM tag morphing. Last accessed: 2017-06.

[QK14] The HDF Group Q. Koziol. Introduction to HDF5 . 08 2014. Last accessed: 2017-05.

List of Figures

1.1 HDF5 Models and Implementations[Gro15a] 6
1.2 Virtual Object Layer(VOL)[Gro15b] 7

2.1 A HDF5 file structure with groups and datasets 11
2.2 Application view of the HDF5 dataset 12
2.3 Creating an HDF5 group 17

4.1 Using the Database VOL Plugin 25
4.2 File 27
4.3 Group 27
4.4 Dataset 28
4.5 Attribute 28
4.6 Open and Create Operations 29
4.7 Primary Data Object Information 29
4.8 Create Operation 30
4.9 Open Operation 30
4.10 Get Operation 30
4.11 Write/Read Operations 31

5.1 Error by open deleted file 39
5.2 Error by open deleted file: Barrier solution 40
5.3 Error by open values that not exists yet 40
5.4 Error by open values that not exists yet: Barrier solution 40

6.1 Create/open file 45
6.2 Create/open group 46
6.3 Create/open dataset 46
6.4 Total I/O Performance 47
6.5 Close Performance 48

List of Listings

2.1 Creating an HDF5 file using read-write modus 15
2.2 Initialization the file for parallel access 16
2.3 VOL class 19
2.4 Group class 20
2.5 Group get function 21
2.6 Enum group get 21

5.1 Create the database connection and the data file 33
5.2 Creation arguments for attribute data 34
5.3 Structure for dataset 34
5.4 Initialization of values in a dataset structure 35
5.5 Part of the Insert operation for the Dataset table 36
5.6 Dataset get callback 37
5.7 Dataset write routine 38
5.8 Attribute read database function 39

6.1 Create the file in parallel 43
6.2 Select hyperslab 43
6.3 Writing dataset by rows 43
6.4 Create memory space 43
6.5 Collective dataset write 43
6.6 Register the Database VOL Plugin 44
6.7 Data and metadata [EB] 48

List of Tables

2.1 The HDF5 Objects 10
2.2 The HDF5 Datatypes 13
2.3 The HDF5 API naming scheme 14
2.4 File Access Modes 15
2.5 Mapping the HDF5 API to the Callbacks 19

6.1 Experiment environment 42

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Wirtschaftsinformatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum	Unterschrift

Veröffentlichung

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik eingestellt wird.

Ort, Datum	Unterschrift