Modified Region Writes

The Core virtual file driver allows the manipulating of HDF5 files in memory instead of in physical storage. In previous versions, changing any part of a file in memory meant the entire file would be written to storage on file close or flush. To improve the performance of the writing to storage operation, a new feature, modified region writes, has been added. With modified region writes, only the changed regions of the file are written to storage.

The intended audience for this feature is advanced users of the Core virtual file driver.

Introduced with

HDF5-1.8.13

May 15, 2014

The HDF Group

For more information about The HDF Group, see www.hdfgroup.org.

Contents

1. Introduction to Modified Region Writes

In the 1.8.13 release of the HDF5 Library, a feature called modified region writes was added to improve the performance of writes to storage. The purpose of this document is to describe the feature and how to use it. The intended audience for this feature is advanced users of the Core virtual file driver (VFD).

The Core (or Memory) VFD allows HDF5 files to be created or opened in memory instead of in physical storage. If an existing file is opened in memory, the entire contents of the file are copied into memory on open. All subsequent manipulations of created or opened files occur in memory. The advantage of working on files in memory is the file operations go much faster, but the disadvantage is significant memory resources may be required when working with large files. On file close or flush, the changes can optionally be propagated to physical storage.

The Core VFD is configured via the following API call:

```
herr_t H5Pset_fapl_core(hid_t fapl_id, size_t increment,
                         hbool_t backing_store)
```

The `backing_store` parameter sets whether or not changes are propagated to physical storage on close. If this parameter is set to 0 (FALSE), then all changes will be lost when the file is closed. If set to 1 (TRUE), then the changes are written to storage on file close or flush. In previous versions of the library when a file was closed, the entire file would be written out if even a single byte has changed. This can be inefficient when very large files are written out after minimal changes have been made.

If files being worked on in memory will be written to disk, the modified region writes feature can be enabled.

1.1. How the Core VFD Tracks File Modifications

When modified region writes are enabled, the Core VFD will track any changes made to the file. On file close or flush, the tracked changes will be written to storage.

As write calls pass through the Core VFD, a list of “start address-end address” pairs representing the writes is updated. This list serves as a map of modified regions in the file. Overlapping or abutting regions are merged as they are inserted into the list.

As a further optimization, a write page size can be set. This feature expands any *dirty regions* (regions with changed bytes) to the nearest page boundaries. Using write pages can minimize seeks and small, inefficient writes when a large number of small non-adjacent writes occur. See the figure below.

Note that these marked regions are at the granularity of the write calls that the library makes. In other words, an entire metadata object or dataset chunk will be marked dirty if even a single byte is changed since the library uses a single write call when metadata objects or dataset chunks are evicted from their respective caches. The Core VFD will make no effort to determine the particular bytes that were modified with respect to the original data.

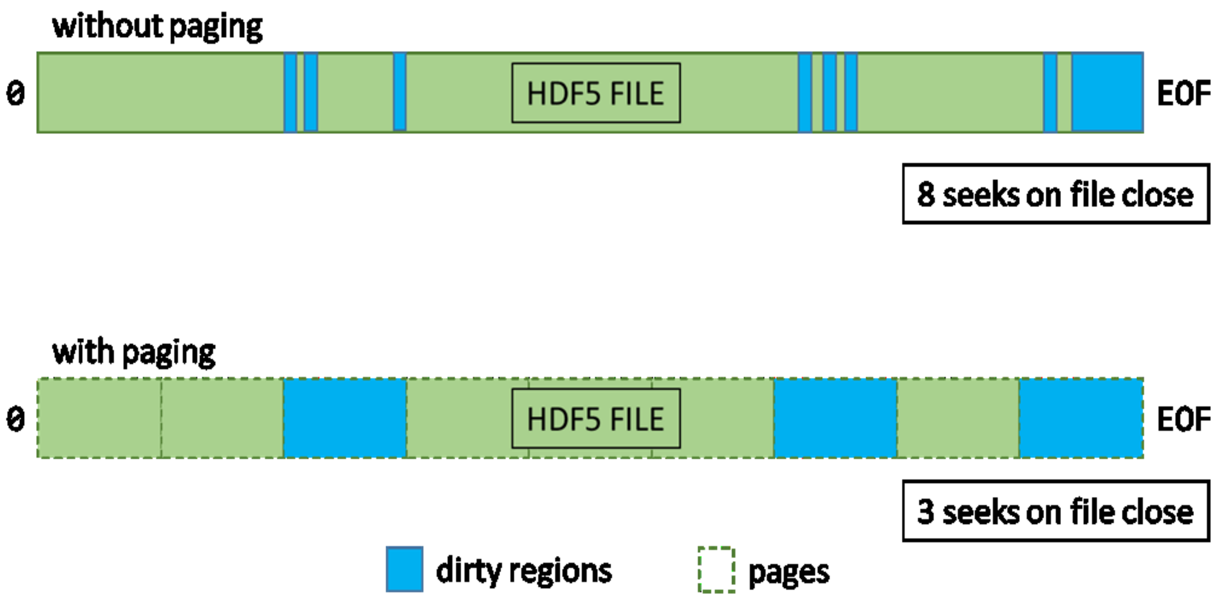

Figure 1. Effect of the paging feature
When the paging feature has been enabled, the in-memory "file" is conceptually divided into multiple pages (dashed lines). Dirtying (modifying) any part of a page marks the entire page as dirty.

1.2. Using the New Feature

The modified region writes feature is turned off by default. Setting the `backing_store` flag to TRUE will not turn modified region writes on.

The modified region writes feature is controlled via the `H5Pget/set_core_write_tracking()` HDF5 API calls. The signatures of these function calls are the following:

```
herr_t H5Pset_core_write_tracking(hid_t fapl_id, hbool_t is_enabled,
                                  size_t page_size)

herr_t H5Pget_core_write_tracking(hid_t fapl_id, hbool_t *is_enabled,
                                  size_t *page_size)
```

Setting the page size to a value greater than 1 turns write tracking on at that page size. Setting a page size of 1 byte disables paging.

More information for these function calls can be found in the *HDF5 Reference Manual*.

1.3. Performance

The performance benefits of the feature will depend heavily on the data access patterns of the application and will have to be evaluated on a case-by-case basis. In cases where the majority of the data would be written out (for example, creating and writing data to a new file), the new feature will likely not impart a significant performance benefit. In cases where a small amount of data will be added or changed (for example, opening an existing file and modifying a small amount of existing data), the performance benefits could be significant.

When performance tuning, the following parameters are likely to have significant effects on I/O throughput:

- The size of the backing store pages (see `H5Pset_core_write_tracking`)
- Dataset layout and chunk size (see `H5Pset_layout` and `H5Pset_chunk`)
- Metadata aggregation size (see `H5Pset_meta_block_size`)
- Using the latest file format (see `H5Pset_libver_bounds`)
- Data layout considerations (arrangement of groups, datasets, and datatypes)

In general, anything that promotes the aggregation of changes made to the file will enhance the performance of this feature. Unfortunately, empirical testing will typically be required to determine the “sweet spot” between reducing the number of seeks and minimizing the amount of data written out.

More information for these function calls can be found in the *HDF5 Reference Manual*.

2. References

For more information, see the entries for the `H5Pset_fapl_core`, `H5Pget_core_write_tracking`, and `H5Pset_core_write_tracking` function calls in the *HDF5 Reference Manual* at http://www.hdfgroup.org/HDF5/doc/RM/RM_H5Front.html.

2.1. The Virtual File Layer and Virtual File Drivers

The HDF5 Library uses a layered architecture. The lowest layer is the virtual file layer (VFL). The VFL handles low-level file I/O via virtual file drivers (VFDs). The VFL is an abstraction layer in the HDF5 Library that maps I/O operations such as "read" to concrete I/O calls like the POSIX `read()` call or the Win32 `ReadFile()` call. Each VFD implements a different I/O scheme: some examples are MPI-I/O, POSIX I/O, and in-memory I/O. This VFL/VFD scheme allows abstract HDF5 file manipulations to be separated from storage I/O operations.

For more information, see "HDF5 Virtual File Layer" at http://www.hdfgroup.org/HDF5/doc/TechNotes/VFL.html.

For more information on virtual file drivers, see the "Alternate File Storage Layouts and Low-level File Drivers" section in "The File" chapter in the *HDF5 User's Guide* at http://www.hdfgroup.org/HDF5/doc/UG/index.html.

3. Revision History

May 5, 2014	Initial version of this document. This document is based on the Core VFD enhancements described in the core CFD paging v5.docx file.