Detecting and Reusing Datatypes in HDF5

James Laird
Quincey Koziol
9/19/05

Introduction to Named Datatypes

Every dataset in HDF5 has a single datatype which is stored on disk as part of the HDF5 file. When there are many datasets (especially many smaller datasets), the space required to store these datatypes can become significant.

Often, HDF5 files include only a few kinds of datatypes, used many times. To save time and disk space and to help prevent mistakes, HDF5 provides a mechanism to reuse a single datatype many times: a Named Datatype. A named datatype is written once and can then be re-used as many times as desired.

Why Have Automatic Creation of Named Datatypes

Named Datatypes are very useful, but must be created and used by hand. If a user doesn't know about or forgets to use them for a given file, that file's size can increase as a datatype is written for each dataset instead of a single named datatype for the entire file. Since the user must create the named datatype separately from the dataset using a different set of API calls, it may seem like more effort than it's worth for datatypes that are only used a few times.

An alternative would be to automate the task of creating and using named datatypes. HDF5 could detect datatypes that were used by more than one dataset and create named datatypes for them automatically. This would ensure that datatypes never took up more disk space than necessary without requiring a user to perform the optimization by hand. It would simplify the library from the user's point of view--they could safely ignore named datatypes without losing storage efficiency.

How Named Datatypes Would Be Handled by the Library

When a user creates a new dataset with a datatype that is not already a named datatype, the HDF5 library searches the file for an identical datatype. If it finds one, it marks the new datatype as being a copy of the already existing datatype rather than creating it from scratch.

This would involve some behind-the-scenes work by the library. Whenever a new datatype was used, the library would need to compare it against every other datatype currently in the file. This would require maintaining a list of such datatypes and writing that list as part of the file. The library would need to track the reference counts on such

datatypes to ensure that they were deleted if and only if the last dataset referencing them was deleted. The most complicated case might be a datatype that was created as part of a dataset, referenced by another dataset, and then had the first dataset deleted--the datatype would be "homeless" but could not be deleted. It might be simplest to consider datatype messages to be independent of their datasets--to make all datatypes named datatypes.

In files with small numbers of datatypes, a simple sorted list of the datatypes used in the file will suffice. However, since this list must be searched every time a new datatype is created to see if this datatype already exists in the file, using a list could result in significant delays in files with many (thousands) of datatypes. Thus, this list of datatypes could "mature" into a more complex data structure (a B-tree, for instance) when there are too many datatypes for a list to handle. The number of datatypes considered "too many" could be adjusted by the user to optimize search time or disk space.

Detecting datatypes automatically might also lend itself to helpful new HDF5 API calls--when creating a new dataset, for instance, users might be able to say "use the same datatype that this dataset does" without having to create a named datatype themselves.

All of the internal detail would be invisible to users, who would simply discover that their HDF5 files didn't take up as much disk space anymore. If they wanted to, they could adjust the number of shared datatypes required before the datatypes were stored in a B-tree instead of a list. The library might also provide functions to query the number of such datatypes being shared, and to query whether any given datatype was shared or not. Naturally, users would still be able to use named datatypes manually.

Drawbacks of Automatically Created Named Datatypes

The most noticable drawback of this automatic detection would be a cost in performance, especially when all existing datatypes must be searched when a new datatype is created to see if the new datatype already exists in the file. Generally, creation of a datatype doesn't require peak performance and most files will have only a few datatypes--dozens, not thousands. Using a more advanced data structure to facilitate searches would help significantly in files with very many datatypes.

The list of datatypes used in the file would need to be stored on disk, slightly increasing the size of the file (this would take very little space; probably only a few KB even for files with many datatypes). For files with shared datatypes, there should be an overall savings of space, but for files that never re-use a datatype or that already use named datatypes there would be a small amount of extra overhead.

If users can adjust the threshold number of datatypes beyond which HDF5 uses a tree instead of a list to sort them, the users could shoot themselves in the foot by choosing unreasonable settings. The default value would be a reasonable, one, however, so users would need to do this deliberately.

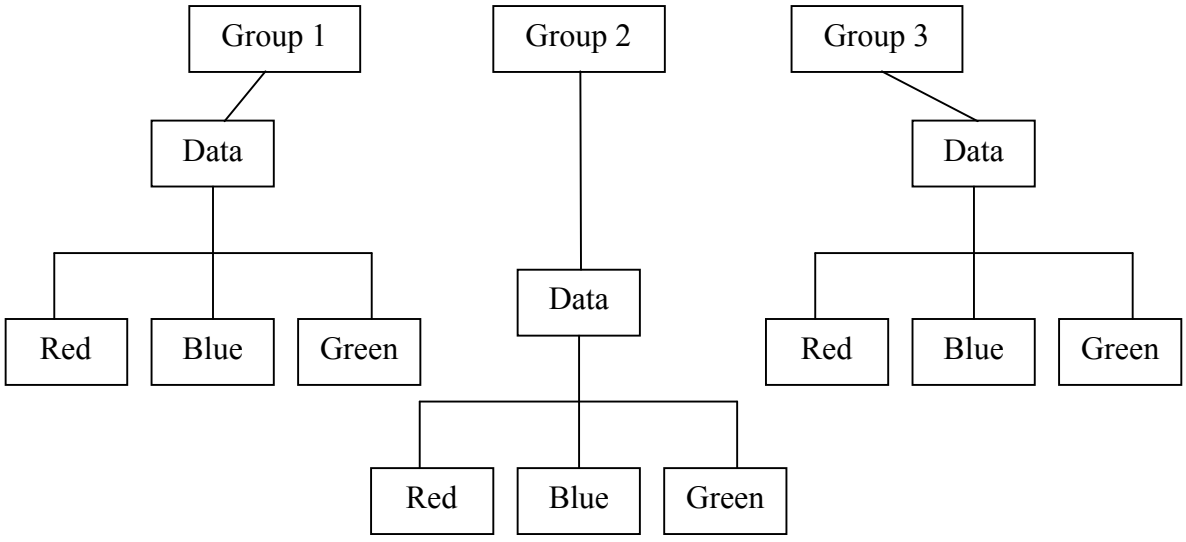
Perceptive users might notice some uneven behavior of the library as the locations of datatypes on disk are different. The order in which creation and deletion of datatypes happens could increase or decrease how much work the library must do and affect where datatypes are stored in the file.

Further Work in the Same Vein

Group Hierarchies

Files often include only a few datatypes which are re-used on many datasets, and a great deal of space can be saved by writing those datatypes to disk only once. Likewise, there may be other duplicated elements in an HDF5 file that could be written to disk once and re-used rather than written once each time they occur.

Hierarchies of groups are one example of this. Files may contain group objects with subgroups and datasets in a regular pattern. For instance, in the following figure, three groups share the layout and names of the groups beneath them.



In this example, HDF5 must store the group name "Data" and the information that it links to datasets named "Red" "Blue" and "Green" three times. If there were 1,000 groups with the same structure, this overhead could become considerable!

Instead, it would make more sense to store this structure of groups, datasets, and links in only once place on disk, and have Group 1, Group 2, and Group 3 refer to it rather than duplicating the information.

The primary question to consider in this case is what information to include in a "group hierarchy." Object names and their link structures are an obvious starting point. Attributes (especially attribute names) may often be duplicated in files, so are probably worth including such a hierarchy. Ultimately, group hierarchies would need to be flexible about which elements are held in common, since different files will be organized differently.

Internally to HDF5, a group hierarchy might consist of an object header or group of object headers that have "common" elements, and then each instance of the hierarchy could "inherit" these elements from the shared hierarchy while supplying any missing information (e.g., addresses of data on disk, extra attributes, etc.). Instances of the hierarchy might also be able to "override" some shared elements locally (e.g., one instance might contain the datasets Red, Blue, and Yellow, overriding the hierarchy's "Green").

In theory, there is no reason that an group hierarchy should not inherit from another group hierarchy, although too many layers of inheritance might degrade performance.

Initially, such hierarchies would probably require users to create and use them manually, since detecting all possible hierarchies in a large file would be quite a challenge!

Objects in these hierarchies might be slower to access than objects outside of hierarchies, since the object header data would reside in multiple places in the file; this should not be a huge delay, especially if the hierarchy information can be stored in the metadata cache. If users are careful in their use of group hierarchies, file sizes should only decrease. However, it would be quite possible for careless users to shoot themselves in the foot and increase their file sizes (by using group hierarchies for only one or two instances, for instance).

Group hierarchies could prove to be very tricky to implement, since there are many ways to alter them:links can be added or removed, objects can be deleted, etc.

Shared Object Headers

In the same vein as shared datatypes and group hierarchies, other elements of metadata might be used in a number of places in the same file. For instance, many datasets in the same file probably have similar dataspaces, filters, and other Dataset Creations Property

List information. Most of this information is stored in a dataset's object header, and the same principle behind named datatypes could be applied to save space and reduce the potential for error.

Users could define a template of shared dataset features to be stored as a named object in an HDF5 file. When a new dataset is created, the user could have it use the values from the stored template rather than entering a new dataspace, filter list, etc. The new dataset would not need to copy these values verbatim, but could simply reference the "named" values already stored in the file. Eventually, named datatypes may be seen as just one example of a larger class of "shared" metadata

The natural form for many of the values that datasets have in common (with the exception of datatypes and dataspaces) is a dataset creation property list. Like the group hierarchy mentioned above, a property list "template" might have values for all of the properties, or only define some and leave each dataset to define the others for itself.

Eventually, these "common" properties could be detected automatically, but like group hierarchies, it will be simpler to implement this in two steps, with the first requiring users to specify which properties will be shared.