DataCutter and A Client Interface for the Storage Resource Broker with DataCutter Services *

Tahsin Kurc†, Michael Beynon†, Alan Sussman†, Joel Saltz†+

† Institute for Advanced Computer Studies
and
Dept. of Computer Science
University of Maryland
College Park, MD 20742

+ Dept. of Pathology
Johns Hopkins Medical Institutions
Baltimore, MD 21287

{kurc,beynon,als,saltz}@cs.umd.edu

Abstract

The continuing increase in the capabilities of high performance computers and continued decreases in the cost of secondary and tertiary storage systems is making it increasingly feasible to generate and archive very large (e.g. petabyte and larger) datasets. Applications are also increasingly likely to make use of archived data obtained by different types of sensors. Such sensors include imaging devices deployed on satellites and aircraft, microscopy related imagery and radiology related imagery.

Simulation or sensor datasets generated or acquired by one group may need to be accessed over a wide-area network by other groups. Datasets frequently describe data associated with collections of very large structured or unstructured grids where each grid point is associated with several variables. Applications frequently need only to obtain portions of a dataset. Required data may correspond to a particular region in a multidimensional space. The application may need to access all data associated in a multidimensional region or it may need only certain variable values at a subsampled set of spatial locations. In addition, in some cases, applications may require data products obtained by aggregating data in one way or another. For instance, a user might require time or space averaged data.

This document describes the design of a middleware infrastructure, called *DataCutter*, that enables subsetting and user-defined filtering of multi-dimensional datasets stored in archival storage systems across a wide-area network. We also describe a client API for Storage Resource

*This research was supported by the National Science Foundation under Grants #ASC-9619020 (UC Subcontract #10152408), and by the Office of Naval Research under Grant #N66001-97-C-8534.

Broker (SRB) clients, which allows SRB clients to carry out subsetting and filtering of datasets stored through the SRB. This API uses a prototype implementation of the DataCutter indexing and filtering services.

Contents

1 Introduction

Many scientific applications generate and use datasets consisting of data values associated with a multi-dimensional space [5, 1, 7]. Scientific simulations typically generate datasets with at least three spatial dimensions and a temporal dimension. Satellite data and microscopy data generally have two (or more) spatial dimensions and a temporal dimension. Applications frequently need to access spatially defined data subsets via a *spatial range query*, which is a multi-dimensional box in the underlying dataset space. Examples of applications that require efficient data subsetting include: (1) programs that explore, compare and possibly visualize results generated by multiple very large scale simulations [7], (2) programs that visualize or generate data products from global coverage satellite data [5], and (3) applications that visualize and classify microscopy data and carry out content based queries that return data subsets [1]. Spatial subsets can encompass contiguous regions of space, as for retrieving satellite data covering a particular geographical region. Spatial subsets can also be defined once features of interest are categorized using spatial indices. For instance, subsetting can be carried out to retrieve simulation data associated with shocks in fluid simulations, or tissue regions with particular cell types in microscopy datasets.

There are various situations in which application-specific non-spatial subsetting and data aggregation can be applied to targeted data subsets. Some data analysis require values for only some of variables at a data point. For example, a computational fluid dynamics simulation dataset can be organized so each data element contains velocity, momentum, and pressure values. An analysis code may only use the pressure value at a grid point, and may ignore values for velocity and momentum. In other cases, there may be a need to obtain an application-dependent low resolution view of a dataset. For example, a hydrodynamics simulation may generate and store flow data (e.g., velocity values) at fine time steps. The analysis may need to be performed using coarser time steps, which requires the stored velocity values to be averaged over several time steps. In these cases, aggregation and transformation operations could be applied to data elements at the data server where they are stored, before returning them to the client where the analysis program is run.

We are developing a middleware infrastructure, called DataCutter [3, 4], to make it possible to explore and analyze scientific datasets stored on archival storage across a wide-area network. DataCutter provides a core set of services, on top of which application developers can implement more application-specific services or combine with existing Grid services such as meta-data management, resource management, and authentication services.

The main design objective in DataCutter is to provide support for accessing subsets of datasets via range queries and for carrying out user-defined aggregations and transformations for very large datasets in archival storage systems, in a shared distributed computing environment. To make efficient use of distributed shared resources within the DataCutter framework, the application processing structure is decomposed into a set of processes, called *filters*. DataCutter uses these

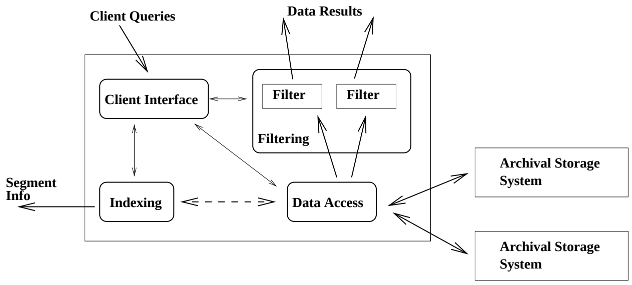

Figure 1: DataCutter system architecture.

distributed processes to carry out a rich set of queries and application specific data transformations. Filters can execute anywhere (e.g., on computational farms), but are intended to run on a machine close (in terms of network connectivity) to the archival storage server or within a proxy close to co-located clients. Filter-based algorithms are designed with predictable resource requirements, which are ideal for carrying out data transformations on shared distributed computational resources. Another goal of DataCutter is to provide common support for subsetting very large datasets through multi-dimensional range queries. Very large datasets may result in a large set of large data files, and thus a large space to index. A single index for such a dataset could be very large and expensive to query and manipulate. To ensure scalability, DataCutter uses a multi-level hierarchical indexing scheme.

The organization of this document is as follows. An overview of the DataCutter architecture is presented in Section 2. Section 3 describes the client API for Storage Resource Broker (SRB) [8] clients. In this section, a metadata format for datasets, indexes and filters in DataCutter also is presented.

2 Overview of the DataCutter Architecture

DataCutter (Figure 1) is being developed as a set of modular services. The client interface service interacts with clients and provides the client API. The data access service provides low level I/O support for accessing the datasets stored on archival storage systems. Both the filtering and indexing services use the data access service to read data and index information from files stored on archival storage systems. The indexing service manages the indices and indexing methods registered with DataCutter. The filtering service manages the filters for application-specific aggregation operations. In the following sections we describe the indexing and filtering services in more detail.

2.1 Indexing

A DataCutter supported dataset consists of a set of data files and a set of index files. Data files contain the data elements of a dataset; data files can be distributed across multiple storage systems.

Each data file is viewed as consisting of a set of *segments*. Each segment consists of one or more data items, and has some associated metadata. The metadata for each segment consists of a minimum bounding rectangle (MBR), and the offset and size of the segment in the file that contains it. Since each data element is associated with a point in an underlying multi-dimensional space, each segment is associated with an MBR in that space, namely a hyperbox that encompasses the points of all the data elements contained in the segment. Spatial indices are built from the MBRs for the segments in a dataset. A segment is the unit of retrieval from archival storage for spatial range queries made through DataCutter. When a spatial range query is submitted, entire segments are retrieved from archival storage, even if the MBR for a particular segment only partially intersects the range query (i.e. only some of the data elements in the segment are requested). DataCutter targets files supported by archival storage systems. From the standpoint of APIs supported by these archival storage systems, a DataCutter segment is a contiguous region of storage – a segment may not actually be stored in a contiguous manner on the host archival storage systems.

One of the goals of DataCutter is to provide support for subsetting very large datasets (sizes up to petabytes). Efficient spatial data structures have been developed for indexing and accessing multi-dimensional datasets, such as R-trees and their variants [2, 6]. However, storing very large datasets may result in a large set of data files, each of which may itself be very large. Therefore a single index for an entire dataset could be very large. Thus, it may be expensive, both in terms of memory space and CPU cycles, to manage the index, and to perform a search to find intersecting segments using a single index file. Assigning an index file for each data file in a dataset could also be expensive because it is then necessary to access all the index files for a given search. To alleviate some of these problems, DataCutter uses a multi-level hierarchical indexing scheme implemented via *summary index files* and *detailed index files* (Figure 2). The elements of a summary index file associate metadata (i.e. an MBR) with one or more segments and/or detailed index files. Detailed index file entries themselves specify one or more segments. Each detailed index file is associated with some set of data files, and stores the index and metadata for all segments in those data files. There are no restrictions on which data files are associated with a particular detailed index file for a dataset. Data files can be organized in an application-specific way into logical groups, and each group can be associated with a detailed index file for better performance. For example, in satellite datasets, each data file may store data for one week. A detailed index file can be associated with data files grouped by month, and a summary index file can contain pointers to detailed index files for the entire range of data in the dataset. DataCutter uses R-trees as its default indexing method.

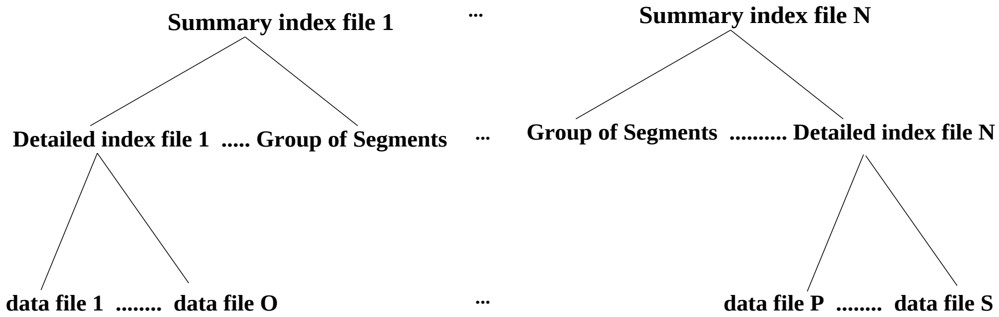

Figure 2: Two-level hierarchical indexing scheme of DataCutter.

2.2 Filtering

The filtering service manages the filters for user-defined filtering/aggregation operations on the data. Filters can be used for a variety of purposes, including elimination of unnecessary data near the data source, pre-processing of segments in a pipelined fashion before sending them to the clients, and data aggregation. Filters can run anywhere, but are intended to run on a machine close (in terms of network connectivity) to the archival storage server or within a proxy. When run close to the archival storage system, filters can reduce the amount of data injected into the network for delivery to the client. In addition, filters can be used to offload some of the required processing from clients to proxies or the data server, thus reducing client workload.

A filter is a user-defined object (e.g., a C++ class object) with methods to carry out application specific processing on the data. A filter consists of an *initialization* function, a *processing* function, and a *finalization* function. The initialization function is executed when the filter is first instantiated by the filtering service. The initialization function is used to create internal state and initialize object member variables. The processing function is run repeatedly as new data arrives at the filter input ports. The finalization function is run when the filter terminates (e.g., by consuming the data on all its input streams).

Communication between a filter and its environment is restricted to its input and output streams. A stream denotes a supply of data to or from the storage media, or a flow of data between two application components, such as between two separate filters or between a filter and a client. Streams deliver data in fixed-size buffers. For example, an input stream can deliver data one segment at a time, and an output stream can output data one segment (after applying the filtering operation) at a time. The sources and sinks for these streams are specified by the client program as a part of installation of the filter. A filter is not allowed to determine (or change) where its input stream comes from or where its output stream goes to. This has two advantages. First, a filter does not handle buffering and scheduling for its communication; the filtering service performs those functions, thereby reducing the complexity of filters. Second, filters can be transparently moved to proxies or other locations as resource constraints at the client and/or server change.

3 Client Interface for the Storage Resource Broker with DataCutter Services

In this section, the client API for Storage Resource Broker (SRB) [8] clients is described. This API has been developed in a joint effort with Mike Wan and Arcot Rajasekar of the SRB group at the San Diego Supercomputing Center (SDSC). The goal is to make it possible for SRB clients to perform spatial subsetting and filtering on data collections accessible through the SRB.

The current API uses a prototype implementation of the indexing and filtering services, which implements a subset of the functionality of DataCutter. In the current version we provide the following functionality:

- Building/deleting indexes on datasets.
- Subsetting a dataset through range queries. A range query is a multi-dimensional bounding box defined in the underlying multi-dimensional space of the dataset. Subsetting is carried out by performing a search (using the index created on the dataset) for data segments that intersect a range query.
- Carrying out user-defined filtering operations on data segments before returning them to the client.
- Adding/registering new filtering functions.

The main restrictions of the current implementation are:

- Users cannot add new index methods. Only the default indexing method of DataCutter, R-tree indexing, can be used.
- The filtering service does not support distributed execution of filters. Users can register multiple filters, but only one filter can be executed at a time for a single query, and only at the SRB server.

Note that SRB/DataCutter is a rapidly evolving system. We plan to gradually integrate more functionality in future releases. Hence, the interface and underlying system are subject to changes in the future. Currently, the interface has only C/C++ language bindings. The API specification in this document assumes that the reader has SRB API specifications available (SRB API manual is available on the Web [8]). This document only describes the interface for SRB clients to use DataCutter services.

3.1 Creating an Index

```
int sfoCreateIndex(srbConn *conn, sfoClass class, int catType, char *hostName,
                   char *inIndexName, char *outIndexName, char *resourceName)
```

Input Parameters

conn SRB server connect information. This can be created by a call to the SRB `clConnect` function. Refer to the SRB manual [8] for more information.

class Subsetting scheme (SFO – search and filter operations). It should be set to `DataCutterCl` for DataCutter operations.

catType SRB catalog type (e.g., MCAT). Refer to the SRB manual [8] for more information about SRB catalog types.

hostName Name of the server host on which to run the create index operation. If it is set to NULL, the operation is executed at the current server to which the client is connected.

inIndexName Name of the SRB collection that contains the input linear index files.

outIndexName Name of the SRB collection where the index files created by the indexing service will be stored.

resourceName SRB resource name. Refer to the SRB manual [8] for more information about SRB resource names.

The `sfoCreateIndex` function creates an index for a set of data files stored in the SRB. Metadata about the data files to be indexed and the linear index files are created by the user in the collection `inIndexName` in the SRB (e.g., using SRB I/O functions). The resulting index files are stored in the collection `outIndexName` in the SRB. If the collection `outIndexName` does not exist, it is created by the indexing service.

Error Codes

`sfoCreateIndex` returns the following values.

`DC_OK` Operation succeeded.

`DC_DATACAT_ERR` Cannot access/read dataset catalog file.

`DC_INDEXCAT_ERR` Cannot access/read index catalog file.

`DC_RTREE_ERR` Cannot create the R-tree index files.

```
# Dataset catalog file
# SRB collection 1
<SRB collection name>
<number of data files in this collection>
<SRB file name>
<SRB file name>
...
<SRB file name>
# SRB collection 2
<SRB collection name>
<number of data files in this collection>
<SRB file name>
<SRB file name>
...
```

Figure 3: The format of the ASCII dataset catalog file.

DC_INTER_ERR Some internal error occurred. DataCutter allocates internal buffers, creates internal state for various operations etc. This error indicates that some error has occurred while performing internal operations.

The user is required to provide three sets of files in the SRB collection specified by `inIndexName`; a *dataset catalog file*, an *index catalog file*, and a set of *linear index files*. The dataset and index catalog files are assigned a well-defined name within the collection (`data.cat` for the dataset catalog file, `index.cat` for the index catalog file) so that the indexing service can access those files. In the next section we describe the format of these files. The DataCutter indexing service expects these files to be binary files. In this document, we describe the format of ASCII files. We provide a utility program to convert the ASCII files into binary files, which is also described.

3.1.1 Dataset Catalog, Index Catalog, and Linear Index Files

A **dataset catalog** contains the path to and the name of each data file in the dataset. Note that files are stored in collections in the SRB (corresponding to directories in a UNIX file system), and are assigned object IDs (corresponding to file names in UNIX). The format of the ASCII dataset catalog file is given in Figure 3. A line starting with "#" is considered a comment and is ignored. An example dataset catalog is shown in Figure 4.

An **index catalog** contains the names of the linear index files. The format of the ASCII index catalog file is given in Figure 5. A line starting with "#" is considered a comment and is ignored.

The **linear index files** contain unordered lists of metadata for the segments in the dataset. The

```
srb_test1_collection
4
test11_dataset
test12_dataset
test13_dataset
test14_dataset
srb_test2_collection
2
test21_dataset
test22_dataset
```

Figure 4: An example dataset catalog file.

```
# Index catalog file
<number of linear index files>
<SRB file name>
<SRB file name>
...
<SRB file name>
```

Figure 5: The format of the ASCII index catalog file.

linear index files correspond to DataCutter *detailed index* files (see Section 2.1), and each linear index file can be associated with one or more data files in the dataset. In the current implementation, the indexing service creates a detailed index file from each linear index file, and a single summary index file for all detailed index files for the dataset. The format of the linear index file is given in Figure 6. In the ASCII linear index file, a line beginning with "#" is considered a comment and is ignored. The format of the binary files for dataset and index catalog files differ from the format of ASCII dataset and index catalog files. However, the format of the binary file for the linear index is the same as the format of the ASCII file, except that there are no comment lines and no white spaces between entries in the binary file. Therefore, the user can directly generate binary linear index files without generating the ASCII files.

<`file type`> describes the file type. It is an `unsigned char` and should be set to `A` for an ASCII file, and `B` for a binary file.

<`dim`> is the number of dimensions of the bounding boxes for the data segments.

<`number of data files`> is the number of data files indexed by this index file.

<`data file id`> is the logical data file id. This value corresponds to the rank of the datafile

```
# Linear index file
<file type>
<dim>
<number of data files>
# data file 1
<data file id>
<number of segments>
# segment 1 of data file 1
<min x> <min y> ... <min k>
<max x> <max y> ... <max k>
<offset> <size>
<min x> <min y> ... <min k>
<max x> <max y> ... <max k>
<offset> <size>
...
<min x> <min y> ... <min k>
<max x> <max y> ... <max k>
<offset> <size>
# data file 2
<data file id>
<number of segments>
# segment 1 of data file 2
<min x> <min y> ... <min k>
<max x> <max y> ... <max k>
<offset> <size>
# segment 1 of data file 2
<min x> <min y> ... <min k>
<max x> <max y> ... <max k>
<offset> <size>
...
```

Figure 6: The format of linear index file.

in the dataset catalog in the order the name of data files appears. For instance, in Figure 4, the `data file id` for "test11_dataset" is 0, and for "test21_dataset" is 4.

<`number of segments`> is the number of segments indexed in the data file by this index file.

<`min x`> <`min y`> `...` <`min k`> are the minimum values in each dimension of the bounding box of the segment. Each value is a `double`.

<`max x`> <`max y`> `...` <`max k`> are the maximum values in each dimension of the bounding box of the segment. Each value is a `double`.

<`offset`> <`size`> are the offset and size of the segment in the data file. Offset and size

```
create_catalogs <command-line parameters>

command-line parameters:
 -l <ASCII linear index file> <binary linear index file>
      create binary linear index file (<binary linear index file>) from
      ASCII linear index file (<ASCII linear index file>). Note that
      the name of the binary linear index file should be the same as
      the name listed in index catalog file.
 -d <ASCII dataset catalog file>
      create binary dataset catalog file (data.cat) from ASCII dataset
      catalog file.
 -i <ASCII index catalog file>
      create binary index catalog file (index.cat) from ASCII index
      catalog file.
 -cd
      print contents of binary dataset catalog file (data.cat) to
      standard output in ASCII dataset catalog file format.
 -ci
      print contents of binary index catalog file (index.cat) to
      standard output in ASCII index catalog file format.
 -cl <binary linear index file>
      print contents of binary linear index file (<binary linear index file>)
      to standard output in ASCII index catalog file format.
 -h
      print this information to standard output
```

Figure 7: The synopsis of `create_catalogs` program.

values are of type `unsigned int`.

As was stated previously, the DataCutter indexing service expects dataset catalog, index catalog and linear index files to be binary files, stored in the `inIndexName` collection in the SRB. The indexing service uses a different format for binary dataset and index catalog files. A utility program, called `create_catalogs`, is provided so that the user can convert ASCII dataset and index catalog files into the binary format used by the indexing service. The same utility program can also be used to convert ASCII linear index files into binary linear index files. This program is located in the `utility` directory in the software distribution. The synopsis of this program is given in Figure 7.

3.2 Deleting an Index

```
int sfoDeleteIndex(srbConn *conn, sfoClass class, int catType,
                   char *hostName, char *indexName)
```

Input Parameters

conn SRB server connect information. Created by a call to the SRB `clConnect` function. Refer to the SRB manual [8] for more information.

class Subsetting scheme (SFO – search and filter operations) class. It should be set to `DataCutterCl` for DataCutter operations.

catType SRB catalog type. Refer to the SRB manual [8] for more information about SRB catalog types.

hostName Name of the server host on which to run the delete index operation. If it is set to NULL, the operation is executed at the current server to which the client is connected.

indexName Name of the collection that contains the index files.

The `sfoDeleteIndex` function deletes an index that was created by a call to `sfoCreateIndex`. The name of the collection that contains the index files is given in `indexName`.

Error Codes

`sfoDeleteIndex` returns the following values.

`DC_OK` Operation succeeded.

`DC_INDEXDEL_ERR` Could not delete the index.

`DC_INTER_ERR` Some internal error occurred. DataCutter allocates internal buffers, creates internal state for various operations etc. This error indicates that some error has occurred while performing internal operations.

3.3 Searching an Index

```
typedef struct {
   int    dim;   /* number of dimensions of the bounding box */
   double *min;  /* array of minimums in each dimension      */
   double *max;  /* array of maximums in each dimension      */
} sfoMBR; /* Bounding box structure */

typedef struct {
   sfoMBR   segmentMBR;      /* bounding box of the segment            */
   char     *objID;          /* object in SRB that contains the segment */
   char     *collectionName; /* collection where object is stored      */
   unsigned int offset;      /* offset of the segment in the object    */
   unsigned int size;        /* size of segment                        */
} segmentInfo; /* segment metadata information */

typedef struct {
   int         segmentCount;  /* number of segments returned          */
   segmentInfo *segments;     /* array of segment metadata information */
   int         continueIndex; /* continuation flag                    */
} indexSearchResult; /* search result structure */

int sfoSearchIndex(srbConn *conn, sfoClass class, char *hostName,
                   char *indexName, void *query,
                   indexSearchResult **myresult, int maxSegCount)
```

Input Parameters

conn SRB server connect information. Refer to the SRB manual [8] for more information.

class Subsetting scheme (SFO – search and filter operations) class. It should be set to `DataCutterCl` for DataCutter operations.

hostName Name of the server host on which to run the search index operation. If it is set to NULL, the operation is executed at the current server to which the client is connected.

indexName Name of the SRB collection that contains the index files.

query Pointer to the query structure. DataCutter queries use the following query structure:

```
typedef struct {
 int    dim;   /* number of dimensions of the query bounding box */
 double *min;  /* array of minimum values in each dimension      */
 double *max;  /* array of maximum values in each dimension      */
} rangeQuery;
```

`dim` is the number of dimensions. `min` is the array of minimum values in each dimension, `max` is the array of maximum values in each dimension of the query.

maxSegCount Maximum length of the segmentInfo array that is returned from a call to `sfoSearchIndex`.

Output Parameters

myresult Stores the index search results. The `myresult` structure is allocated by the SRB. The allocated space can be deallocated by a call to

```
sfoFreeIndexResults(indexSearchResult *myresult)
```

The `sfoSearchIndex` function performs a search on `indexName` to find the segments that intersect the range query given in the `query` parameter. The metadata for segments is returned in the output parameter `myresult`. The segment metadata is stored in the `segments` array in a `segmentInfo` structure. The `segments` array is allocated by the indexing service– the calling program is responsible for freeing the segment array. The `segmentCount` ($\leq$ `maxSegCount`) stores the number of entries returned in the `segments` array, which may be less than `maxSegCount`. The returned `continueIndex` field is greater than zero if there are more results to return; -1 is returned if there are no more results. The DataCutter indexing service maintains internal state for the query so that subsequent calls to `sfoGetMoreSearchResult` (described next) can return more results. If there are no more results to return, the indexing service deletes the internal query state and sets the `continueIndex` field to -1. `maxSegCount` limits the maximum number of segments to be returned from a single call to the `sfoSearchIndex` function.

Error Codes

`sfoCreateIndex` returns the following values.

`DC_OK` Operation succeeded.

`DC_DATACAT_ERR` Cannot read dataset catalog file.

`DC_INDEXCAT_ERR` Cannot read index catalog file.

`DC_RTREE_ERR` Cannot access the R-tree index files.

`DC_QUERY_ERR` The number of query dimensions does not match the number of dataset dimensions.

`DC_SEGINFO_ERR` Cannot create segment information data structures. For example, allocation of a buffer to hold segment information may have failed.

`DC_INTER_ERR` Some internal error occurred. DataCutter allocates internal buffers, creates internal state for various operations etc. This error indicates that some error has occurred while performing internal operations.

```
int sfoGetMoreSearchResult(srbConn *conn, sfoClass class, char *hostName,
                           int continueCond, indexSearchResult **myresult,
                           int maxSegCount)
```

Input Parameters

conn SRB server connect information. Refer to the SRB manual [8] for more information.

class Subsetting scheme (SFO – search and filter operations) class. It should be set to `DataCutterCl` for DataCutter operations.

hostName Name of the server host on which to run the get more result operation. If it is set to NULL, the operation is executed at the current server to which the client is connected.

continueCond If set to a value greater than 0, return more results. If it is set to -1, return no more results.

maxSegCount Maximum number of segment information to be returned from a call to `sfoGetMoreSearchResult`.

Output Parameters

myresult Stores the index search results. The `myresult` structure is allocated by SRB. The allocated space can be deallocated by a call to

```
sfoFreeIndexResults(indexSearchResult *myresult)
```

The `sfoGetMoreSearchResult` function is used to get more results from a search operation for a query. This function should be called after a call to the `sfoSearchIndex` function. The `continueCond` parameter should be set greater than zero to get more results. If it is set to -1, no more results are returned for the query. The DataCutter indexing service maintains internal state for the query submitted via a call to `sfoSearchIndex`. If the input parameter `continueCond` is set to -1, then the indexing service deletes the internal state for the query, and no results are returned. If the state of the current query is deleted, `sfoSearchIndex` should be called again to start a new search. The value of the `maxSegCount` parameter limits the length of the segment metadata array that can be returned from a call to the `sfoGetMoreSearchResult` function, as for `sfoSearchIndex`.

Note that in the current implementation, neither `sfoSearchIndex` nor `sfoGetMoreSearchResult` returns a query handle. DataCutter maintains internal state for the current query. Thus, only one query can be submitted at a time by any one client. A call to `sfoSearchIndex` will delete the state for the current query and initialize the internal state for the new query, and subsequent calls to `sfoGetMoreSearchResult` will return results for the new query. We plan to extend the API and the underlying system in future releases so that a client can have multiple simultaneous active queries.

Error Codes

`sfoGetMoreSearchResult` returns the following values.

`DC_OK` Operation succeeded.

`DC_RTREE_ERR` Cannot access the R-tree index files.

`DC_QUERY_ERR` The query dimensions does not match the dataset dimensions.

`DC_SEGINFO_ERR` Cannot create segment information data structures. For example, allocation of a buffer to hold segment information may have failed.

`DC_NOTINIT_ERR` This error indicates that `sfoGetMoreSearchResult` has been called before a call to `sfoSearchIndex`.

`DC_INTER_ERR` Some internal error occurred. DataCutter allocates internal buffers, creates internal state for various operations etc. This error indicates that some error has occurred while performing internal operations.

3.4 Applying a Filter

```
typedef struct {
   segmentInfo segInfo;  /* info on segment data buffer after filter oper. */
   char        *segment; /* segment data buffer after filter is applied    */
} segmentData;
typedef struct {
    int         segmentDataCount; /* #segments in segmentData array */
    segmentData *segments;        /* segmentData array */
    int         continueIndex;    /* continuation flag */
} filterDataResult;
int sfoApplyFilter(srbConn *conn, sfoClass class, char *hostName,
                   int filterID, char *filterArg, int numOfInputSegments,
                   segmentInfo *inputSegments, filterDataResult **myresult,
                   int maxSegCount)
```

Input Parameters

conn SRB server connect information. Refer to the SRB manual [8] for more information.

class Subsetting scheme (SFO – search and filter operations) class. It should be set to `DataCutterCl` for DataCutter operations.

hostName Name of the server host on which to run the filter operation. If it is set to NULL, the filter operation is executed at the current server to which the client is connected.

filterID An integer value denoting the id of the filter to be applied. To retrieve data segments without applying a filter, it should be set to `-1`.

filterArg User arguments to the filter function. The filter argument is a string, which is parsed by the filter function.

numOfInputSegments Number of segments on which the filter operation is to be applied.

inputSegments Array of metadata for input segments, on which the filter operation is to be applied.

maxSegCount Maximum number of segments that may be returned from a call to the `sfoApplyFilter` function.

Output Parameters

myresult Array of segments produced by the filter operation. The `myresult` structure is allocated by SRB. The allocated space can be deallocated by a call to

```
sfoFreeFilterResults(filterDataResult *myresult)
```

The `sfoApplyFilter` function applies a filter operation, denoted by `filterID`, to the set of segments, whose metadata is given in `inputSegments`. Results are returned in the output parameter `myresult`. The value of `continueIndex` in the `filterDataResult` structure is set greater than or equal to zero if there are more segments to be processed. Its value is set to -1 if there are no more segments. As for the indexing service, the filtering service also maintains internal state about the list of segments to be processed, so that subsequent calls to `sfoGetMoreFilterResult` can process the remaining segments in the list of input segments. If there are no more segments to be processed, the internal state is deleted, and the returned `continueIndex` is set to -1.

Error Codes

`sfoApplyFilter` returns the following values.

`DC_OK` Operation succeeded.

`DC_FILTERID_ERR` The value of `filterID` is incorrect.

`DC_FILTER_ERR` Error in carrying out user-defined filter operations.

`DC_SEGMENT_ERR` Cannot read or process a segment in the list of segments. For example, allocation for a buffer to hold data for a segment may have failed. The `segment` field in `segmentData` is set to NULL for those segments.

`DC_INTER_ERR` Some internal error has occurred. DataCutter allocates internal buffers, creates internal state for various operations etc. This error indicates that some error has occurred while performing internal operations.

```
int sfoGetMoreFilterResult(srbConn *conn, sfoClass class, char *hostName,
                           int continueCond, filterDataResult **myresult,
                           int maxSegCount)
```

Input Parameters

conn SRB server connect information. Refer to the SRB manual [8] for more information.

class Subsetting scheme (SFO – search and filter operations) class. It should be set to `DataCutterCl` for DataCutter operations.

hostName Name of the server host on which to run the filter operation. If it is set to NULL, the filter operation is executed at the current server to which the client is connected.

continueCont If set greater than 0, return more results. If it is set to -1, return no more results.

maxSegCount Maximum number of segments that should be returned from each call to the `sfoGetMoreFilterResult` function.

Output Parameters

myresult Array of segments produced by the filter operation. The `myresult` structure is allocated by SRB. The allocated space can be deallocated by a call to

```
sfoFreeFilterResults(filterDataResult *myresult)
```

The `sfoGetMoreFilterResult` function is used to get more filtered segments, from the list of segments passed to the `sfoApplyFilter` function. The `continueCond` parameter should be set greater than zero to get more results. If it is set to -1, the function does not return more results. The semantics of `sfoGetMoreFilterResult` is similar to that of `sfoGetMoreSearchResult` in the indexing service. If the input parameter `continueCond` is set to -1, the filtering service deletes its internal state for the current list of segments, and no results are returned. If the current state is deleted, `sfoApplyFilter` should be called again with a new list of segments to apply a filter operation. The value of the `maxSegCount` parameter limits the number of the segments that can be returned from a call to the `sfoGetMoreFilterResult` function.

Note that in the current implementation, neither `sfoApplyFilter` nor `sfoGetMoreFilterResult` returns a handle for the current list of segments to which the filter operation is applied. DataCutter maintains internal state for the list of segments. Thus, only one list of segments can be processed at a time for any one client. A call to `sfoApplyFilter` will delete the internal state for the current list of segments and initialize the internal state for the new list. Subsequent calls to `sfoGetMoreFilterResult` will return results for the new list of segments.

Error Codes

`sfoGetMoreFilterResult` returns the following values.

`DC_OK` Operation succeeded.

`DC_NOTINIT_ERR` This error indicates that `sfoGetMoreFilterResult` has been called before a call to `sfoApplyFilter`.

`DC_FILTER_ERR` Error in carrying out user-defined filter operations.

`DC_SEGMENT_ERR` Error in reading and/or processing a segment. For example, allocation for a buffer to hold data for a segment may have failed. The `segment` field in `segmentData` is set to NULL for those segments.

`DC_INTER_ERR` Some internal error has occurred. DataCutter allocates internal buffers, creates internal state for various operations etc. This error indicates that some error has occurred while performing internal operations.

```
#include "sfoDCFilterImpl.h"

class dcFilterImpl {
public:
  dcFilterImpl();
  virtual ~dcFilterImpl();

  /* initialize the filter */
  virtual int init(char *user_arg);

  /* Methods to disclose memory requirements for scratch */
  /* space and the output segment data                    */
  virtual int  scratchSpaceSize(void) { return -1; };
  virtual void setScratchSpace(void *scratchBufferPtr) { };
  virtual int  outputSegmentSize(void) { return -1; };

  /* The filter operation to be applied. */
  virtual int process(void *segBufferIn, segmentInfo  *segInfoIn,
                      segmentData *segOut);

  /* finalization */
  virtual int finalize(void);
};
```

Figure 8: The filter base class for adding new filters.

3.5 Implementing a New Filter Function

The filtering service in the SRB/DataCutter system is implemented in C++. Adding new filters to the system is achieved via C++ inheritance and virtual methods. The filtering service provides a base class, which the user subclasses and then implements the virtual methods. The definition of the base class in the prototype implementation is given in Figure 8:

init This method is used to initialize class data members from the user argument (`filter_arg`) passed in from `sfoApplyFilter`. The user has to implement this method. `init` must return 0 if the operation succeeds, and -1 otherwise.

scratchSpaceSize This method is used to disclose the amount of scratch memory space (in bytes) to be allocated by the filtering service. If this method returns a value greater than 0, the filtering service allocates a buffer of that size. After the buffer is allocated, the `setScratchSpace` method (described next) is called by the filtering service to pass the pointer to the buffer to the user code. The buffer may be used in the user-defined `process` method as scratch space, for example to hold intermediate values to process an input segment. The user is not required to implement this method; a default implementation, which returns -1, is provided by the filtering service. In that case, no scratch space is pre-allocated for the filter by the filtering service and user-defined methods are required to dynamically allocate scratch space.

setScratchSpace This method is used to pass the pointer to the scratch buffer, allocated by the filtering service, to the user code. The `scratchBufferPtr` parameter is a pointer to the buffer allocated by the filtering service. The user may implement this method to initialize internal pointers to access the scratch space buffer in other methods. It is a NULL pointer if the `scratchSpaceSize` method returns -1.

outputSegmentSize This method is used to disclose the (maximum) size (in bytes) of the buffer required to hold the resulting segment after the filtering operation is applied to an input segment. If this method returns a value greater than 0, the filtering service allocates a buffer, the size of which is equal to the value returned from the method, to be used to hold an output segment generated in the `process` method. The user does not have to implement this method; a default implementation, which returns -1, is provided by the filtering service. In that case, no buffer is pre-allocated for the output segments and the user-defined `process` method is required to dynamically allocate space for each output segment.

process This method applies the filter operation on the input segment, `segBufferIn`, with associated metadata `segInfoIn`, and stores the result in `segOut` (see Section 3.4 for the definition of the `segmentData` structure, and Section 3.3 for the definition of the `segmentInfo` structure). `segOut->segInfo` should contain metadata for the output segment after the filter is applied. The `segmentMBR` field of `segInfo` can contain the minimum bounding rectangle for the output segment. The `size` field **must** contain the size (in bytes) of the output segment. The data for the output segment should be stored in `segOut->segment`. Note that the buffer for `segOut->segment` is allocated by the filtering service if the user-defined `outputSegmentSize` returns a value greater than 0. Otherwise, `segOut->segment` will be a NULL pointer, in which case the buffer for the output segment should be allocated in the `process` method. The `size` and `segment` fields are also used by the filtering service to return the output segment data to the client from the SRB server. The user has to implement this method. The **process** method must return 0 if the operation succeeds, and -1 otherwise.

finalize This method is used to free resources allocated by the filter. The user has to implement this method. It must return 0 if the operation succeeds, and -1 otherwise.

The filtering service first calls the **init** method so that the user-defined filter can parse the user argument and perform initialization of member variables, etc. Then, **scratchSpaceSize** and **outputSegmentSize** methods are called by the filtering service to get the size of the buffer space required for the scratch space and output segments. Note that the current filtering service neither enforces user-defined filters to pre-disclose dynamic memory requirements for scratch space and output segments, nor prohibits a filter from using dynamic memory allocation[1] If the **scratchSpaceSize** and **outputSegmentSize** methods return values greater than 0, the filtering service allocates the requested buffer space on behalf of the user-defined filter. The pointer to the buffer for scratch space is passed to the user-defined code through the **setScratchSpace** method. The filtering service calls this function and passes the pointer to the buffer via `scratchBufferPtr` parameter. The user-code in **setScratchSpace** may initialize the internal member pointers to access the buffer space in the **process** method. The **process** method is called for each input segment retrieved from archival storage. The **segOut->segment** field of the **segmentData** structure points to the buffer space (if it has been allocated by the filtering service) to store the output segment after applying the filter operation to the current input segment. If the buffer space is not allocated by the filtering service, the **segOut->segment** is a NULL pointer. In that case, the buffer for the output segment must be allocated dynamically in the **process** method. After all segments are processed, the filtering service calls the **finalize** method so that the user-defined filter can free the resources it allocated.

Currently, adding and registering a new filter function is done by the SRB system administrator. That is, the user should send the source code for a user-defined filter function, implemented as a subclass of **dcFilterImpl**, to the SRB system administrator. The system administrator integrates the new filter class into the SRB server (i.e. compiles the source code and links the class object to the SRB executable), and assigns a **filter id** for the new filter function. This filter id can be used in a call to the **sfoApplyFilter** function, as the value of the **filterID** parameter.

3.6 A Metadata Format for Datasets, Indexes and Filters

In this section, we present a metadata format to describe datasets, indexes and filters registered in SRB/DataCutter system. This metadata format is designed to make use of existing MCAT system so that users can access metadata information through SRB/MCAT client API.

3.6.1 Datasets

The metadata attributes for a dataset consist of `primary-data-info`, `data-class-info`, and `structured-data-info` tables. The primary-data-info keeps the primary metadata information

[1] One of our current research projects is investigating when pre-disclosing resource requirements will be important, how it can be used for developing strategies for better resource management and scheduling of filters for efficient execution in a distributed environment with shared resources, and the implications of pre-disclosing resource requirements in developing applications using filters. We plan to integrate the strategies developed in our research work in future releases.

about the dataset:

data_id The dataset id assigned by the MCAT. It is of type `integer`.

data_name A user defined name for the dataset. It is of type `string`.

data_desc A short text description of the dataset, e.g., "AVHRR Satellite data over Northern America". It is of type `string`.

data_collection A dataset is stored in an SRB collection. This field holds the name of the collection. It is of type `string`.

data_type A user defined type for the dataset, e.g., "Image", "Microscopy", "AVHRR", etc. The type can be used to designate datasets with the same segment structure. A filter designed to process data segments from a dataset can be applied to other datasets with the same data segment structure. It is of type `string`.

The `data-class-info` table keeps the information about the class of the dataset. In general, datasets stored through the SRB can also be accessed by a *search-and-filter class* other than DataCutter (e.g., a geographic information system). The class information is used to categorize a dataset according to the *search-and-filter operations* applicable to the dataset.

class_name The name of the class the dataset belongs to. For example, for datasets that are intended to be accessed through DataCutter, the value of this attribute should be `DataCutterCl`. It is of type `integer`.

The `structured-data-info` table is used to keep application-specific information for datasets. For example, DataCutter datasets are expected to be multi-dimensional. The dimensions and ranges in each dimension can be stored with the `structured-data-info`. The attributes of `structured-data-info` are:

structure_type The type of the structured information. The user-defined type value is used by the client program to correctly decode the internals of the structures. For example, if structured data is stored as an XML file, then the value could be "XML". It is of type `string`.

structure_comments A short description of the structured information. It is of type `string`.

structure_data_id points to the `data_id` that stores the structure. For example, the structure can be an XML file stored in SRB. Then, the `data_id` value can be used to find the name of the collection, where the XML file is stored, from the `primary-data-info` attributes. It is of type `integer`.

internal_structure can be used to store a small amount of structured information in MCAT. It is of type `string`.

Note that there exists a cross-reference in MCAT mapping a *data_name* to the `structured-data-info` table so that a user can access this information through a search on the dataset name.

3.6.2 Indexes

In MCAT, indexes are considered to be a dataset of a particular type. Hence, primary information on indexes (i.e. `primary-data-info`) is already stored as part of the metadata for datasets. The extra information required for indexes describes which index applies to which dataset and the location of the indexes. This information is stored in the `index-data-info` table, with the following attributes:

index_id This is actually the `data_id` in the `primary-data-info` table. It is of type `integer`. The `index_id` is used to find the corresponding set of attributes for the name of the index, description of the index, etc., stored in the `primary-data-info` table.

indexeddata_id The `data_id` of the dataset indexed by this index. It is of type `integer`.

indexeddata_type This value holds information about the type of the index. It allows users to search for specific types of index, e.g., R-tree indexes. It is of type `string`.

index_location The location of the index. An index can be stored in an SRB collection (as for DataCutter), or, for example, can be stored in a relational database (perhaps for another search-and-filter class). It is of type `string`.

index_comments A short description of the index. It is of type `string`.

3.6.3 Filters

In MCAT, filters are considered to be methods, and methods are considered to be datasets of a particular type. Therefore, the primary information about filters can be stored in the `primary-data-info` table in MCAT. The `structured-data-info` table can be used to hold filter specific information, for example the input parameters to a filter. The extra metadata information required for a filter is:

method_id This is actually the `data_id` in the `primary-data-info` table. It is of type `integer`.

data_id The `data_id` of the dataset to which this filter can be applied. It is of type `integer`.

data_type The type of the data to which this segment can be applied. It is of type `string`.

Acknowledgements

We would like to thank Michael Wan and Arcot Rajasekar of the SRB team at San Diego Supercomputer Center for their invaluable help in designing the SRB/DataCutter interface and in integrating DataCutter services into the SRB.

References

[1] Asmara Afework, Michael D. Beynon, Fabian Bustamante, Angelo Demarzo, Renato Ferreira, Robert Miller, Mark Silberman, Joel Saltz, Alan Sussman, and Hubert Tsang. Digital dynamic telepathology - the Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*. American Medical Informatics Association, November 1998.

[2] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of SIGMOD'90*, pages 322–331. ACM Press, May 1990.

[3] Michael D. Beynon, Renato Ferreira, Tahsin Kurc, Alan Sussman, and Joel Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In *Proceedings of the Eighth Goddard Conference on Mass Storage Systems and Technologies/17th IEEE Symposium on Mass Storage Systems*, pages 119–133, College Park, MD, March 2000. IEEE Computer Society Press.

[4] Michael D. Beynon, Tahsin Kurc, Alan Sussman, and Joel Saltz. Design of a framework for data-intensive wide-area applications. Technical Report CS-TR-4104 and UMIACS-TR-2000-04, University of Maryland, Department of Computer Science and UMIACS, February 2000. To appear in the Proceedings of the 2000 Heterogeneous Computing Workshop.

[5] Chialin Chang, Bongki Moon, Anurag Acharya, Carter Shock, Alan Sussman, and Joel Saltz. Titan: A high performance remote-sensing database. In *Proceedings of the 1997 International Conference on Data Engineering*, pages 375–384. IEEE Computer Society Press, April 1997.

[6] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of SIGMOD'84*, pages 47–57. ACM Press, May 1984.

[7] Tahsin M. Kurc, Alan Sussman, and Joel Saltz. Coupling multiple simulations via a high performance customizable database system. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1999.

[8] SRB: The Storage Resource Broker. *http://www.npaci.edu/DICE/SRB/index.html*.