DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage Systems [*]

Michael Beynon[†], Renato Ferreira[†], Tahsin Kurc[†], Alan Sussman[†], Joel Saltz[†‡]

[†] Department of Computer Science
University of Maryland
College Park, MD 20742

[‡] Department of Pathology
Johns Hopkins Medical Institutions
Baltimore, MD 21287

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

Abstract

In this paper we present a middleware infrastructure, called DataCutter, that enables processing of scientific datasets stored in archival storage systems across a wide-area network. DataCutter provides support for subsetting of datasets through multi-dimensional range queries, and application specific aggregation on scientific datasets stored in an archival storage system. We also present experimental results from a prototype implementation.

1 Introduction

Increasingly powerful computers have made it possible for computational scientists and engineers to model physical phenomena in great detail. As a result, overwhelming amounts of data are being generated by scientific and engineering simulations. In addition, large amounts of data are being gathered by sensors of various sorts, attached to devices such as satellites and microscopes. The primary goal of generating data through large scale simulations or sensors is to better understand the causes and effects of physical phenomena. Thus, the exploration and analysis of large datasets plays an increasingly important role in many domains of scientific research. Simulation or sensor datasets generated or acquired by one group may need to be accessed over a wide-area network by other groups. Software support is needed to allow users to obtain needed subsets of very large, remotely stored datasets.

We present a middleware infrastructure, called DataCutter, that enables processing of scientific datasets stored in archival storage systems across a wide-area network. DataCutter provides support for subsetting of datasets through multi-dimensional range queries, and application specific aggregation on scientific datasets stored in an archival storage system. We discuss an implementation of the Virtual Microscope application [2] using DataCutter. The Virtual Microscope is representative of data-intensive applications that involve browsing and processing large multi-dimensional datasets. Other examples include satellite data processing systems [7] and water contamination studies that couple multiple simulators [20]. We also provide experimental performance results for a prototype implementation.

[*] This research was supported by the National Science Foundation under Grant #ACI-9619020 (UC Subcontract # 10152408), and the Office of Naval Research under Grant #N6600197C8534.

2 Motivation and Overview

Over the past several years we have been actively working on data intensive applications that employ large-scale scientific datasets, including applications that explore, compare, and visualize results generated by large scale simulations [20], visualize and generate data products from global coverage satellite data [7], and visualize and analyze digitized microscopy images [2]. Many scientific applications generate and use datasets consisting of data values associated with a multi-dimensional space. 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. 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.

In some cases data analysis can be employed in a collaborative environment, where co-located clients access the same datasets and perform similar processing. For instance, a large group of students in medical training may need to simultaneously explore the same set of digitized microscopy slides, or visualize the same MRI and CT datasets. There may be a large number of overlapping regions of interest, and common processing requirements (e.g., same magnification level for microscopy images, or same transfer functions to convert scalar values into color values) among the users employing the analysis tools (*clients* of the data server). In these cases, caching reused dataset portions closer to the clients (i.e., on the same local area network) can provide significant performance benefits.

We have developed the Active Data Repository (ADR) [6] framework to use for developing parallel applications that make use of large centralized scientific datasets. ADR provides support for accessing subsets of multi-dimensional scientific datasets via range queries, and allows users to integrate user-defined processing of large centralized datasets with storage and retrieval on distributed memory parallel machines with multiple disks attached to each node. ADR is designed as a set of *customizable* and *internal* services. Through the use of customizable services, users can specify and implement application specific dataset

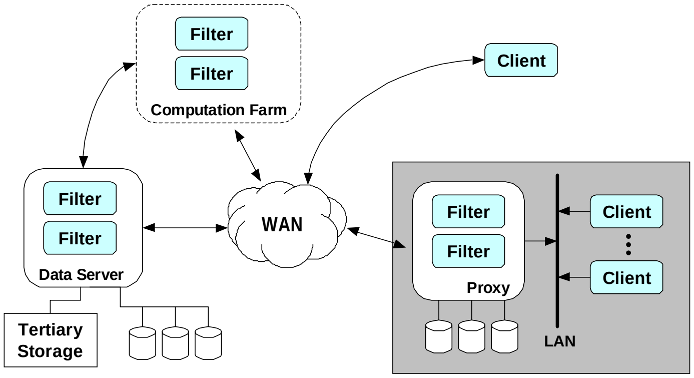


Figure 1: Architecture of the data management/manipulation framework.

indexing, and user-defined aggregation and transformation operations used in processing one or more datasets. The internal services provide support for common operations such as memory management, data retrieval, management of multiple datasets, and query planning and scheduling of processing on a parallel machine. A number of applications have been developed using ADR and good performance has been demonstrated [6, 20]. However, the continuing increase in the capabilities of high performance computers and sensor devices implies that datasets with sizes up to petabytes will be common in the near future. Such vast amounts of data require the use of archival storage systems distributed across a wide-area network. Data analysis, on the other hand, is usually performed on machines at an application scientist's local institution. Efficient storage, retrieval and processing of multiple large scientific datasets on remote archival storage systems is therefore one of the major challenges that needs to be addressed for efficient exploration and analysis of these datasets.

There is a large body of hardware and software research on archival storage systems, including distributed parallel storage systems [19], file systems [23], image servers [22], and data warehouses [18]. Several research projects have focused on digital libraries and geographic information systems [3, 14] that access collections of archival storage systems, high-performance I/O systems [9] and remote I/O [11, 21]. In addition to many end-point solutions, the *Grid* [8, 10, 13] has been emerging in recent years as infrastructure to link distributed computational, network and storage resources, and to provide services for unified, secure, efficient and reliable access. Several research projects have focused on providing services in a Grid environment, such as Globus [12], which provides services to access computational resources, and the Storage Resource Broker (SRB) [21], which provides uniform UNIX-like I/O interfaces and meta-data management services to access collections of distributed data resources. However, providing support for efficient exploration and processing of very large scientific datasets stored in archival storage systems in a Grid environment remains a challenging research issue, and the necessity of infrastructure to provide such support was recognized in recent Grid forums [16].

We are developing an infrastructure to make it possible to explore and analyze scientific datasets stored on archival storage across a wide-area network. Figure 1 illustrates the framework architecture. It consists of two major components: *DataCutters* and *Proxies*. A proxy provides support for caching and management of data near a set of clients. The goal is to reduce the response time seen by a client, decrease the amount of redundant data transferred across the wide-area network, and improve the scalability of data servers. Our prior work on proxies can be found in [5].

The new middleware infrastructure, called DataCutter, provides support for processing of scientific datasets stored in archival storage systems in 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. Our main design objective in DataCutter is to extend and apply the salient features of ADR (i.e. support for accessing subsets of datasets via range queries and user-defined aggregations and transformations) for very large datasets in archival storage systems, in a shared distributed computing environment. In ADR, data processing is performed where the data is stored (i.e. at the data server). In a Grid environment, however, it may not always be feasible to perform data processing at the server, for several reasons. First, resources at a server (e.g., memory, disk space, processors) may be shared by many other competing users, thus it may not be efficient and cost-effective to perform all processing at the server. Second, datasets may be stored on distributed collections of storage systems, so that accessing data from a centralized server may be very expensive. Moreover, distributed collections of shared computational and storage systems can provide a more powerful and cost-effective environment than a centralized server, if they can be used effectively. Therefore, 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 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 (see Figure 1). Filter-based algorithms are designed with predictable resource requirements, which are ideal for carrying out data transformations on shared distributed computational resources.

Many filter-based algorithms were originally developed and analyzed by our group for Active Disks [1, 24]. These filter-based algorithms carry out a variety of data transformations that arise in earth science applications and applications of standard relational database sort, select and join operations. In the DataCutter framework we are extending these algorithms and investigating the application of filters and the stream-based programming model in a Grid environment.

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. In the following sections we describe the DataCutter infrastructure, in particular the indexing and filtering services, and present an implementation of the Virtual Microscope [2] using DataCutter.

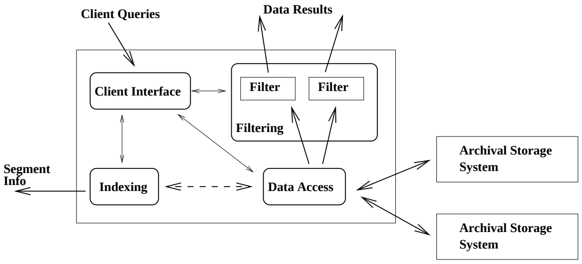


Figure 2: DataCutter system architecture.

3 DataCutter

The architecture of DataCutter (Figure 2) is being developed as a set of modular services. The client interface service interacts with clients and receives multi-dimensional range queries from them. 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.

3.1 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).

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 [4]. 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*. 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. However, the infrastructure allows users to add new indices and indexing methods (through the use of C++ class inheritance).

3.2 Filters

In DataCutter, *filters* are used to perform non-spatial subsetting and data aggregation. Filters are managed by the filtering service. A filter is a specialized user program that pre-processes data segments retrieved from archival storage before returning them to the requesting client. 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 are executed in a restricted environment to control and contain their resource consumption. Filters can execute anywhere[1], but are intended to run on a machine close (in terms of network connectivity) to the archival storage server or within a proxy (see Figure 1). When run close to the archival storage system, filters may reduce the amount of data injected into the network for delivery to the client. Filters can also be used to offload some of the required processing from clients to proxies or the data server, thus reducing client workload.

Filters are written in a *stream-based programming model*, originally developed for programming Active Disks [1]. A filter consists of an initialization function, a processing function, and a finalization function. The initialization function is run when the filter is first installed on the data server. The processing function is run repeatedly as new data arrives at the filter input ports (via streams). The finalization function is run when the filter terminates (either by consuming the data on all its input streams or by calling `exit`).

The programming model for filters is built around the notion of a stream abstraction. 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 can be of two types – file streams and pipe streams. File streams are a sequence of ranges in files, and constitute the primary access method for data residing in secondary or archival storage. Pipe streams are a representation of a unidirectional flow of data between any two components of the application, and are used for both control interaction and data transfer. The stream-based programming model provides and enforces a standard interface for accessing streams [1]. Streams deliver data in fixed-size buffers whose size is fixed at

[1] Filters do not migrate state, and are not written in a platform independent language such as Java, but rather are compiled for the target platform and placed by the DataCutter filter service at runtime.

the time the stream is created. The size of the stream buffer cannot be changed after its creation. A filter may, optionally, contain scratch space, which is allocated on its behalf before it is initialized and is automatically reclaimed after it exits. Filters specifically cannot dynamically allocate and deallocate space, which allows the filtering service to better perform scheduling and buffer management and enable execution in environments with limited resources (e.g., memory).

Communication between a filter and its environment is restricted to its input and output streams. The sources and sinks for these streams are specified by the client program as a part of filter installation. A filter cannot 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 need to handle buffering and scheduling for its own communication, thereby reducing the complexity of filters. Second, filters can be transparently executed in proxies or other convenient locations as resource constraints at the client and/or server change.

4 An Example: the Virtual Microscope using DataCutter

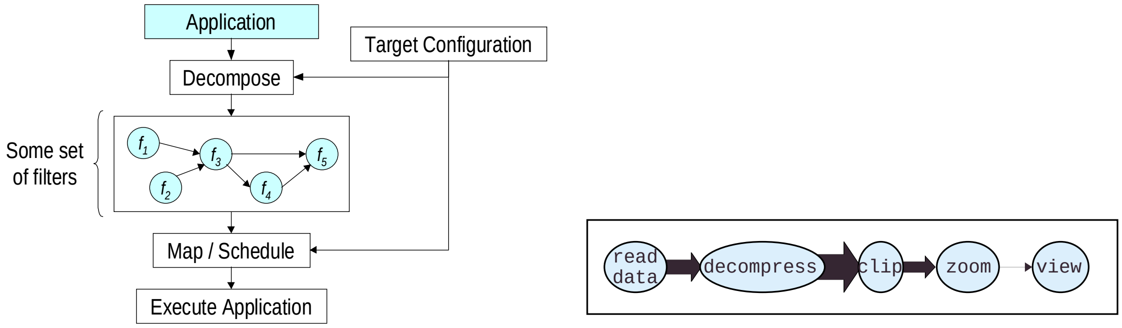
In this section we describe an implementation of the Virtual Microscope application [2] using the DataCutter infrastructure.

4.1 The Virtual Microscope

The Virtual Microscope is a client-server software system, which is designed to realistically emulate a high power light microscope. The data used by the Virtual Microscope are digitized images of full microscope slides at high power. Digitized images from a slide effectively form a three-dimensional (3D) dataset because each slide may contain multiple focal planes, each of which is a 2D image. Images are stored at the highest magnification level, and the size of a single slide typically varies from $100MB$ to $5GB$, compressed. At a basic level, the system is required to provide interactive response times similar to a physical microscope, including continuously moving the stage and changing magnification. A typical query allows a client to request a 2D rectangular region at a particular magnification from within the bounds of a single focal plane. The processing for the query requires projecting high resolution data onto a grid of suitable resolution (governed by the desired magnification) and appropriately compositing pixels that map to a single grid point, to avoid introducing spurious artifacts into the displayed image. The Virtual Microscope can support completely digital dynamic telepathology [2], as well as enabling new modes of operation that cannot be achieved with a physical microscope, such as simultaneous viewing and manipulation of a single slide by multiple users.

4.2 The Original Implementation

The original Virtual Microscope system is composed of two components; a client to generate queries and display the results (i.e. images), and a server, implemented with the Active Data Repository, to process the queries. A protocol has been defined between the client and the server for exchanging queries and results. The server is composed of a frontend and a backend. The frontend interacts with clients; it receives queries from clients and forwards



(a) Overall process (b) Virtual Microscope filters

Figure 3: Process of applying filter and stream-based programming model.

them to the backend. The backend consists of one or more processes (when run on a parallel machine). The processing of a query is carried out entirely in the backend.

In order to achieve high I/O bandwidth, each focal plane in a slide is regularly partitioned into data chunks, each of which is a rectangular subregion of the 2D image. When the host machine is a parallel machine with multiple disks attached to each processor, data chunks are declustered across all the disks to achieve I/O parallelism. Each pixel in a chunk is associated with a coordinate (in x- and y-dimensions) in the entire image. As a result, each data chunk is associated with a minimum bounding rectangle (MBR), which encompasses coordinates of all the pixels in the chunk. An index is created using the MBR of each chunk. Since the image is regularly partitioned into rectangular regions, a simple lookup table, consisting of a 2-dimensional array, serves as an index.

During query processing, the backend process finds the chunks that intersect the query region, and reads them from the local disks. In the original server implementation, each data chunk is stored in compressed form (JPEG format). Hence, each retrieved chunk is first *decompressed*. Then, it is *clipped* to the query region. Afterwards, each clipped chunk is *subsampled* to achieve the magnification (*zoom*) level given in the query. The resulting image blocks are directly sent to the client. The client *viewer* assembles and displays the image blocks from each of the backend processes to form the query output.

4.3 An Implementation using DataCutter

Developing an application with the DataCutter infrastructure requires partitioning of a dataset used by the application into segments, and building spatial indices on the segments. DataCutter provides default interfaces to create and search R-trees. In this implementation of the Virtual Microscope, we employed the data chunks in the original implementation as the segments, and used the default R-tree indexing method of DataCutter. Each focal plane consists of a partitioned set of data files, each with a single detailed index file, and one summary index file is created to index all focal planes in a slide.

The next step in developing the application is to implement the application specific processing using filters and the stream-based programming model. Figure 3(a) illustrates the general steps for implementing an application using filters. First, the application processing structure is decomposed into a set of filters. An important issue is how to choose the number of filters for implementing the application processing. For instance, the original server im-

```
VM_zoom::init() {
  // Allocate from pre-allocated scratch space
  bufOut = AllocFromScratch(getOutputStreamBufferSize());
}
VM_zoom::process(stream_t &st) {
  DC_StreamBuffer *buf;
  VMQuery *query;
  VMChunk *chunk;

  // receive and extract the query
  buf = st.ins[0].read();
  query = VMUnpackQuery(buf);

  // while more data to read from input stream
  while ((buf = st.ins[1].read()) != NULL) {
    // extract chunk and perform zoom
    chunk = VMUnpackChunk(buf);
    zoom_chunk(chunk, query);

    // pack into buffer and write to output stream
    bufOut = VMPackChunk(chunk);
    st.outs[0].write(&bufOut);
    FreeToScratch(chunk->Data);
  }
}
VM_zoom::finalize() {
  FreeToScratch(bufOut);
}
```

```
void VM_zoom::zoom_chunk(VMChunk *chunk,
                          VMQuery *query) {
  int rel_zoom = query->Zoom/chunk->Zoom;
  int width = chunk->Width/rel_zoom;
  int height = chunk->Height/rel_zoom;
  int size = width*height*PIXELSIZE;

  char *pSrc = chunk->Data;
  char *pDst = chunk->Data = AllocFromScratch(size);
  // subsample the image block
  for (j = height; j>0; --j) {
    for (i = width; i>0; --i) {
      memcpy(pDst, pSrc, PIXELSIZE);
      pSrc += rel_zoom*PIXELSIZE;
      pDst += PIXELSIZE;
    }
    pSrc += rel_zoom*chunk.Width*PIXELSIZE;
  }
  // update chunk metadata
  chunk->Zoom = query->Zoom;
}
```

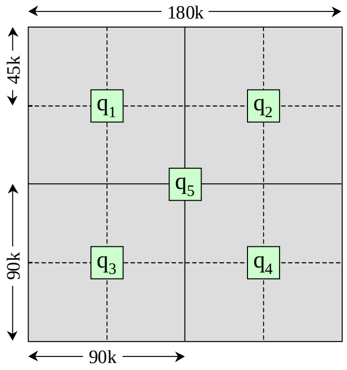
Figure 4: Zoom filter pseudo-code, which performs subsampling of an image chunk based on the query magnification (zoom).

plementation could be considered a single filter. In choosing the appropriate decomposition, we need to consider the complete data flow path from data generation to ultimate consumption as well as the target machine configuration, which can be a distributed collection of heterogeneous machines. The main goal is to achieve efficient use of limited resources in a distributed and heterogeneous environment. We are currently developing techniques and guidelines to assist in this important step.

The selected decomposition of the Virtual Microscope system into filters is shown in Figure 3(b). The figure only depicts the main dataflow path of image data through the system; other streams related to the client-server protocol are not shown for clarity. In this implementation each of the main processing steps in the server is a filter:

- **read-data:** Each full-resolution rectangular image block (i.e. data chunk) that intersects the query window is read from disk, and immediately written to the output stream before the next read operation.
- **decompress:** Image blocks are read individually from the input stream. The block is decompressed using JPEG decompression and converted into a 3 byte RGB format, and the block's metadata header is changed to indicate the image block's new format. The image block is then written to the output stream.
- **clip:** Uncompressed image blocks are read from the input stream. Portions of the block that lie outside the query region are removed, and the clipped image block is written to the output stream.
- **zoom:** Image blocks are read from the input stream, one block at a time. Using the requested magnification in the query, image blocks are subsampled to achieve the desired magnification. The resulting image block is written to the output stream.
- **view:** Image blocks are received for a given query, collected into a reply required for the Java client, and sent to the client.

Figure 4 illustrates the high-level code for the zoom filter. Implementation of the filters is done through C++ class inheritance and virtual functions. The DataCutter infrastructure



Filter	*Total Volume*	*Volume Per Chunk*
read data	3.60 MB	102.52 KB
decompress	83.42 MB	2373.04 KB
clip	57.83 MB	1645.02 KB
zoom	.90 MB	25.70 KB

Figure 5: The 2-dimensional dataset and queries used in the experiments. The solid and dashed lines show different partitionings of the dataset into files for the experiments. The table shows transmitted sizes for q_5.

provides base classes and virtual functions for *initialization*, *processing*, and *finalization* operations in filters, as well as functions to set scratch space size and stream buffer size (not shown in the figure). The zoom filter has two input streams and one output stream. It reads the query from stream 0 (st.ins[0]) and data from stream 1 (st.ins[1]), and subsamples the received data chunks using the `zoom_chunk` function. The zoom filter uses scratch space to store the results during subsampling and to pack the subsampled chunk into the output buffer. The result is written to the output stream (st.outs[0]), which connects the filters *zoom* and *view*.

As is discussed in Section 3, streams between filters deliver the data in individual fixed-size buffers. In the current implementation we send data chunks in stream buffers, and the size of the buffer is chosen to be the maximum size of a chunk in the dataset. This allows us to reuse code from the original Virtual Microscope implementation with little modification.

5 Experimental Results

We have developed a prototype implementation of the DataCutter services. Using this prototype, we have implemented a simple data server for digitized microscopy images stored on the IBM HPSS system [17] at the University of Maryland. The implementation is based on the functionality of the Virtual Microscope and uses the filters described in Section 4.

Our HPSS setup has 10TB of tape storage space and 500GB of disk cache, and is accessed through a 10-node IBM SP with 4 multiprocessor (1 4-processor and 3 2-processor) and 6 single processor nodes. In all of the experiments, we use a 4GB 2D image dataset, in JPEG compressed format (90GB uncompressed), created by stitching together small digitized microscopy images. This dataset is equivalent to a digitized slide with a single focal plane that has $180K \times 180K$ RGB pixels. The 2D image is regularly partitioned into 200×200 data segments and stored in the HPSS as a set of files. For all experiments we use 5 different queries, each of which covers 5×5 segments of the image (see Figure 5). Execution times presented in this section are the response time seen by the visualization client (including submitting a query and receiving the results) and are the average of processing each query 5 times. One node of the IBM SP is used to run the indexing service, and the client was run on a SUN workstation connected to the SP node through the depart-

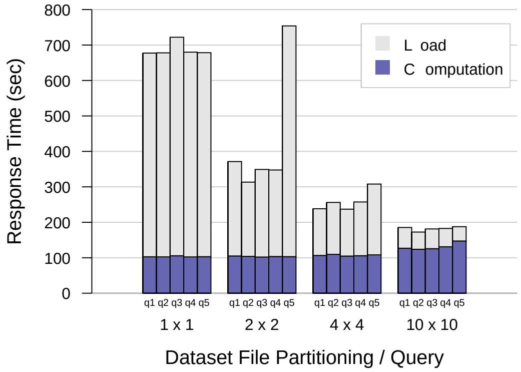
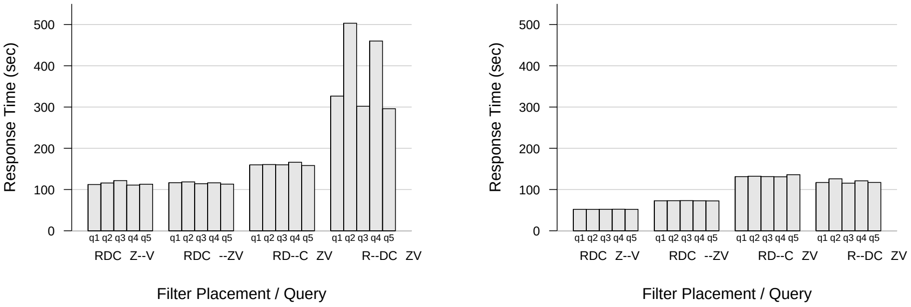


Figure 6: Query execution time with the dataset organized into 1×1, 2×2, 4×4 and 10×10 files. *Load* shows the time to open and access the files, which contain segments that intersect a query. *Computation* shows the sum of the execution time in the indexing service, for searching segments that intersect a query, and in the filtering service, for processing the retrieved data via filters. All filters and the indexing service were run on the same SP node.

ment Ethernet. We experimented with different placements of the filters by running some of the filters (and the filtering service) on the same SP node where the indexing service is executed, as well as on the SUN workstation where the client is run.

The first experiment isolates the impact of organizing the dataset into multiple files. Figure 6 shows the results when the 2D image is partitioned into 1×1, 2×2, 4×4 and 10×10 rectangular regions, and all data segments in each region are stored in a data file. Figure 5 illustrates the partitioning of the dataset into 1×1 (entire rectangle), 2×2 (solid lines), and 4×4 (dashed lines) files. Each data file is associated with a *detailed index* file, and there is one *summary index* file for all the detailed index files for each partitioning. As is seen in the figure, the *load* time decreases as the number of files is increased. This is because of the fact that HPSS loads the entire file onto disks used as the HPSS cache when a file is opened. When there is a single file, the entire 4GB file is accessed from HPSS for each of the queries– in these experiments, all data files are purged from disk cache after each query is processed. When the number of files increases, only a subset of the detailed index files and data files are accessed using the multi-level hierarchical indexing scheme, decreasing the time to access data segments. Note that the *load* time for query 5 for the 2×2 case is substantially larger than that of other queries, because query 5 intersects segments from each of the four files (Figure 5), hence the same volume of data is loaded into the disk cache as in the 1×1 case. The load time for that query is also larger than that in 1×1 case because of the overhead of seeking/loading/opening four files instead of a single file. The *computation* time, on the other hand, remains almost the same, except for the 10×10 case, where it slightly increases. Each query intersects more files as the dataset is partitioned more finely. As a result, the overhead from opening and accessing a large number of detailed index files can increase the computation time. These results, demonstrate that applications can take advantage of the multi-level hierarchical indexing scheme by organizing a dataset into an appropriate set of files. However, having too many files may increase computation time, potentially decreasing overall efficiency when multiple similar queries are executed on the same dataset.



(a) no zooming (no subsampling)

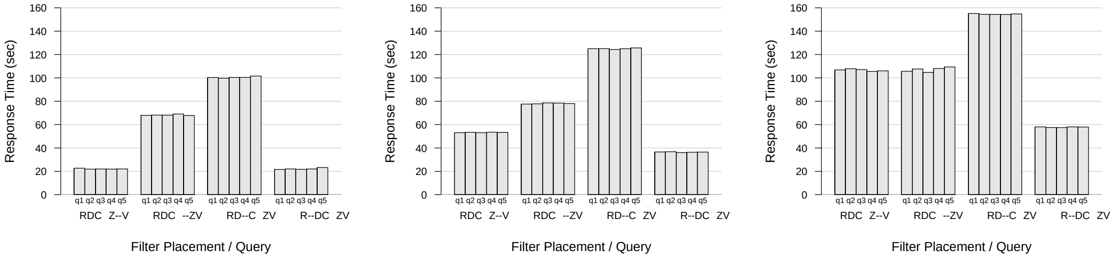
(b) subsampling by a factor of 8

Figure 7: Execution time of queries under varying zoom (subsampling) factor. R,D,C,Z,V denote the filters *read_data*, *decompress*, *clip*, *zoom*, and *view* respectively. {server}–{client} denotes the placement of the filters in each set.

Next, we consider varying the placement of the filters under different conditions. Figures 7 and 8 show query execution times for different filter placements under varying processing requirements (i.e. the subsampling factor) and varying server load, where the server is the machine where the *read_data* filter is run. The different server loads in Figure 8 were emulated by artificially slowing down the set of filters running on the server machine. Figure 7 shows the query execution times when the image is viewed at the highest magnification (no subsampling) and when the subsampling factor is 8 (i.e. only every 8th pixel in each dimension is displayed). As is seen from the figure, when there is no subsampling, query execution times remain almost the same whether the zoom filter is run at the server or at the client, because the volume of data transfer between server and the client is the same in both cases. When queries require subsampling, the placement of the zoom filter affects performance, since the volume of data sent from the server to the client decreases if the zoom filter is executed at the server. As is also seen from the figure, running the filters at the server (RDCZ-V) achieves better performance than running them at the client (R-DCZV) as would be expected since the client is a less powerful machine than the server.

Figure 8 shows query execution times when the server load changes. As is seen in the figure, as the server load increases (or the client becomes faster), running the filters on the client machine achieves better performance. The experimental results show that the decomposition of an application processing structure into filters and placement of the filters are important factors that affect overall performance. One of our long term goals in this work is to devise methodologies for a wide range of data-intensive applications for efficient restructuring of application processing structure into filters with the stream-based programming model, as well as developing cost models for filters to achieve efficient execution under changing processing requirements and system resource availability.

The query execution times for the original optimized Virtual Microscope server versus the prototype filter implementation using DataCutter are shown in Figure 9. In this experiment the entire dataset is loaded from HPSS and stored on a single local disk on a SUN Ultra 1 workstation since the original server is implemented to access datasets stored on disks. The loading of the dataset took 4750 seconds (1 hour 19 minutes). The original server is run as a single process, and all filters in the DataCutter implementation are executed on the same SUN workstation where the dataset is stored. In both cases the client is run on an-



(a) 1x server load (b) 2x server load (c) 4x server load

Figure 8: Execution time of queries under varying server load. $1\times$ server denotes the case where the server is dedicated to running the filters, whereas $2\times$ and $4\times$ increased load implies server execution time doubles and quadruples that of the dedicated case. The subsampling factor is 8 in all cases.

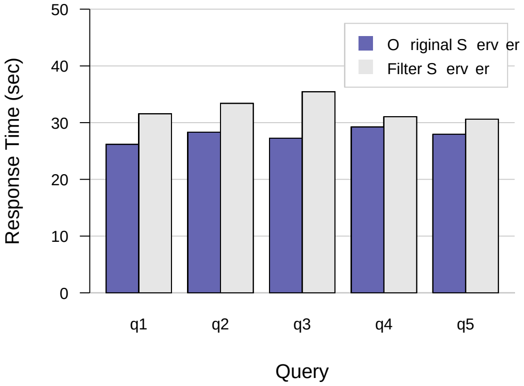


Figure 9: Query execution times for the original server and the server implemented using the DataCutter with filters. The subsampling factor is 8 in all queries.

other SUN Ultra 1 workstation connected to the local Ethernet segment. As is seen from the figure, the filter implementation does not introduce much overhead compared to the optimized original server. The percent increase in query execution time ranges from 6% to 30% across all queries. We should note that the timings do not include the time for loading the dataset, which can substantially increase for larger datasets and datasets stored in archival storage systems across a wide-area network. In addition, the use of filters in DataCutter takes advantage of pipelining and threaded execution, especially when the filters are run on multiprocessor architectures, resulting in overall higher performance.

6 Conclusions and Future Work

In this paper we have presented a middleware infrastructure, called DataCutter, to provide support for processing of large datasets stored in archival storage systems in a wide-area network environment. DataCutter provides support for subsetting of very large datasets through spatial range queries via hierarchical multi-level indexing, and user-defined aggregation and transformation on datasets via filters. We are in the process of developing standard interfaces and a client API for the DataCutter services. We also have several active projects that involve the use of DataCutter services and proxies. In a joint project with the data intensive computing environments group at the San Diego Supercomputing Center, we are interfacing DataCutter with the Storage Resource Broker (SRB) [21]. Our goal is to make it possible for SRB clients to perform spatial subsetting and data aggregation on distributed data collections accessible through the SRB. In a project with The University of Maryland Global Land Cover Facility [15], we are integrating DataCutter and the Active Data Repository (ADR) with the GLCF data servers to make it possible to visualize and generate data products from Landsat Thematic Mapper (TM) datasets stored in HPSS. We will extend our proxy infrastructure to cache data on disks as well as in memory, and integrate proxies with ADR so that clients can generate data products using ADR and data cached on the disks. DataCutter will provide support for accessing subsets of TM datasets from HPSS. We also plan to work on developing distributed stream-based algorithms via use of filters and carry out performance studies for a wider range of data intensive applications.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 81–91. ACM Press, Oct. 1998. ACM SIGPLAN Notices, Vol. 33, No. 11.

[2] A. Afework, M. D. Beynon, F. Bustamante, A. Demarzo, R. Ferreira, R. Miller, M. Silberman, J. Saltz, A. Sussman, and H. Tsang. Digital dynamic telepathology - the Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*. American Medical Informatics Association, Nov. 1998.

[3] Alexandria Digital Library. *http://alexandria.ucsb.edu/*.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM-SIGMOD Conference*, pages 322–331, Atlantic City, NJ, May 1990.

[5] M. Beynon, A. Sussman, and J. Saltz. Performance impact of proxies in data intensive client-server applications. In *Proceedings of the 1999 International Conference on Supercomputing*. ACM Press, June 1999.

[6] C. Chang, R. Ferreira, A. Sussman, and J. Saltz. Infrastructure for building parallel database systems for multi-dimensional data. In *Proceedings of the Second Merged IPPS/SPDP Symposiums*. IEEE Computer Society Press, Apr. 1999.

[7] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. 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, Apr. 1997.

[8] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The Data Grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *http://www.globus.org/*, 1999.

[9] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. PASSION: Parallel and scalable software for input-output. Technical Report SCCS-636, NPAC, Sept. 1994. Also available as CRPC Report CRPC-TR94483.

[10] I. Foster. The Beta Grid: A national infrastructure for computer systems research. In *NetStore'99*, 1999.

[11] I. Foster, D. K. Jr., R. Krishnaiyer, and J. Mogill. Remote I/O: fast access to distant storage. In *Fifth Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, pages 14–25. ACM Press, 1997.

[12] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[13] I. Foster and C. Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1999.

[14] Geographic Information Systems. *http://www.usgs.gov/research/gis/title.html*.

[15] The University of Maryland Global Land Cover Facility. *http://glcf.umiacs.umd.edu*.

[16] Grid Forum. Birds-of-a-Feather Session, SC99, Nov 1999.

[17] The High Performance Storage System (HPSS). *http://www.sdsc.edu/hpss/hpss1.html*.

[18] T. Johnson. An architecture for using tertiary storage in a data warehouse. In *the Sixth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies, Fifteenth IEEE Symposium on Mass Storage Systems*, 1998.

[19] W. E. Johnston and B. Tierney. A distributed parallel storage architecture and its potential application within EOSDIS. In *NASA Mass Storage Symposium*, Mar. 1995.

[20] T. M. Kurc, A. Sussman, and J. 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, Mar. 1999.

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

[22] N. Talagala, S. Asami, and D. Patterson. The Berkeley-San Francisco fine arts image database. In *the Sixth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies, Fifteenth IEEE Symposium on Mass Storage Systems*, 1998.

[23] M. Teller and P. Rutherford. Petabyte file systems based on tertiary storage. In *the Sixth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies, Fifteenth IEEE Symposium on Mass Storage Systems*, 1998.

[24] M. Uysal. *Programming Model, Algorithms, and Performance Evaluation of Active Disks*. PhD thesis, Department of Computer Science, University of Maryland, College Park, 1999.