The Active Data Repository
Version 0.9

Chialin Chang, Tahsin Kurc, Alan Sussman, Joel Saltz
Department of Computer Science
University of Maryland
College Park, MD 20742
{chialin,kurc,als,saltz}@cs.umd.edu

February 10, 2001

For additional information and documents on the Active Data Repository
http://www.cs.umd.edu/projects/adr

Contents

1 Introduction

Analysis and processing of very large multi-dimensional scientific datasets (i.e. where data items are associated with points in a multi-dimensional attribute space) is an important component of science and engineering. Examples of very large datasets include long running simulations of time-dependent phenomena that periodically generate snapshots of their state (e.g. hydrodynamics and chemical transport simulation for estimating pollution impact on water bodies [4, 7, 14, 13], magnetohydrodynamics simulation of planetary magnetospheres [20], simulation of a flame sweeping through a volume [18], airplane wake simulations [15]), archives of raw and processed remote sensing data (e.g. AVHRR [17], Thematic Mapper [22], MODIS [16]), and archives of medical images (e.g., high resolution light microscopy [2], CT imaging, MRI, sonography). For example, a dataset of coarse-grained satellite data (with 4.4 km pixels), covering the whole earth surface and captured over a relatively short period of time (10 days) is about 4.1GB; a finer-grained version (1.1 km per pixel) contains about 65 GB of sensor data. In medical imaging, the size of a single digitized composite slide image at high power from a light microscope is over 7GB (uncompressed), and a single large hospital can process thousands of slides per day.

An increasing number of applications make use of very large multi-dimensional scientific datasets. These applications have several important characteristics. Both the input and the output are often disk-resident datasets. Applications may use only a subset of all the data available in input and output datasets. Access to data items can be described by a *range query*, namely a multi-dimensional bounding box in the underlying multi-dimensional attribute space of the dataset. Only the data items whose associated coordinates fall within the multi-dimensional box are retrieved. The processing structures of these applications also share common characteristics. Figure 1 shows high-level pseudo-code for the basic processing loop in these applications. The processing steps consist of retrieving input and output data items that intersect the range query (steps 1–2 and 4–5), mapping the coordinates of the retrieved input items to the corresponding output items (step 6), and aggregating, in some way, all retrieved input items that map to the same output data items (steps 7–8). Correctness of the output usually does not depend on the order input data items are aggregated. That is, the aggregation operation is an *associative* and *commutative* operation– the aggregation functions allowed correspond to the *distributive* and *algebraic* aggregation functions defined by Gray et. al [11]. The mapping function, $Map(i_e)$, maps an input item to a set of output items. An intermediate data structure, referred to as an *accumulator*, is used to hold intermediate results during processing. For example, an accumulator can be used to keep a running sum for an averaging operation. The aggregation function, $Aggregate(i_e, a_e)$, aggregates the value of an input item with the current intermediate result stored in the accumulator element (a_e). The output dataset from a query is usually much smaller than the input dataset, hence steps 4–8 are called the *reduction* phase of the processing. Accumulator elements are allocated and initialized (step 3) before the reduction phase. Note that step 2 is necessary only if the output dataset already exists and output elements are needed to initialize accumulator elements. The intermediate results stored in the accumulator are post-processed to produce final results (steps 9–11).

Some typical examples of applications that make use of multi-dimensional scientific datasets are satellite data processing applications [1, 6, 19], the Virtual Microscope and analysis of microscopy data [2, 10], and simulation systems for water contamination studies [13]. In satellite data processing, for example, earth scientists study the earth by processing remotely-sensed data continuously acquired from satellite-based sensors. Each sensor reading is associated with a position (latitude and longitude) and the time the reading was recorded. In a typical analysis [1, 19], a range query defines a bounding box that covers a part or all of the earth surface over a period of time. Data

```
    O ← Output Dataset, I ← Input Dataset
    [S_I, S_O] ← Intersect(I, O, R_query)
    (* Initialization *)
1.  foreach o_e in S_O do
2.    read o_e
3.    a_e ← Initialize(o_e)
    (* Reduction *)
4.  foreach i_e in S_I do
5.    read i_e
6.    S_A ← Map(i_e) ∩ S_O
7.    foreach a_e in S_A do
8.      a_e ← Aggregate(i_e, a_e)
    (* Output *)
9.  foreach a_e do
10.   o_e ← Output(a_e)
11.   write o_e
```

Figure 1: The basic processing loop in the target applications. Here, R_{query} denotes the range query, S_I is the subset of the input that intersects the range query, and S_O is the subset of the output that intersects the range query.

items retrieved from one or more datasets are processed to generate one or more composite images of the area under study. Generating a composite image requires projection of the selected area of the earth onto a two-dimensional grid [23]; each pixel in the composite image is computed by selecting the "best" sensor value that maps to the associated grid point. Another example is the *Virtual Microscope* [2, 10], which supports the ability to interactively view and process digitized data from tissue specimens. The raw data for such a system can be captured by digitally scanning collections of full microscope slides under high power. The digitized images from a slide are effectively a three-dimensional dataset, since each slide can contain multiple two-dimensional focal planes. At the basic level, a range query selects a region on a focal plane in a slide. The processing for the Virtual Microscope requires projecting high resolution data onto a grid of suitable resolution (governed by the desired magnification) and appropriately compositing pixels that map onto a single grid point, to avoid introducing spurious artifacts into the displayed image.

This document describes the Active Data Repository (ADR) [5], an infrastructure that integrates storage, retrieval and processing of large multi-dimensional datasets on distributed memory parallel architectures with multiple disks attached to each node. ADR targets applications with the processing structure shown in Figure 1. ADR is designed as a set of modular services, implemented in C++. Through use of these services, ADR allows customization for application specific processing (i.e. the *Initialize*, *Map*, *Aggregate*, and *Output* functions), while providing support for common operations such as memory management, data retrieval, and scheduling of processing across a parallel machine. The system architecture of ADR consists of a front-end and a parallel back-end. The front-end interacts with clients, and forwards range queries with references to user-defined processing functions to the parallel back-end. During query execution, back-end nodes retrieve input data and perform user-defined operations over the data items retrieved to generate

the output products. Output products can be returned from the back-end nodes to the requesting client, or stored in ADR.

This document is organized as follows. Section 2 presents an overview of the Active Data Repository, and the services it provides. The ADR front-end services are described in Section 3. Section 4 describes the ADR back-end, and customizable services in greater detail. The definitions of the base classes for each service are also described in that section. Loading datasets into ADR, along with a description of the support provided by the ADR, is discussed in Section 5. Section 6 describes how to install ADR, how to register datasets and customized implementations of the services in ADR. Appendix A describes some of the utility classes provided by ADR that can be used in the customization of back-end services. Finally, the implementation of an example application is described in Appendix B.

2 Overview of the Active Data Repository

Figure 2 shows the architecture of an application implemented as a customized ADR instance. The full application suite consists of one or more *clients*, a *front-end*, and a *back-end*. A client program, implemented for a specific application or application domain, generates requests into ADR, and post-processes the results from the back-end. The front-end translates the requests into ADR queries, and sends them to the back-end for processing, where the datasets are stored.

2.1 Datasets in ADR

A dataset loaded into ADR is partitioned into a set of chunks, each of which consists of one or more data items. A chunk is the unit of I/O and communication in ADR. That is a chunk is retrieved as a whole during query processing. Every data item in a dataset is associated with a point in the multi-dimensional attribute space underlying the dataset. Similarly, every chunk is associated with a minimum bounding rectangle (MBR) that encompasses the coordinates (in the associated attribute space) of all items in the chunk. ADR assumes a distributed-memory multicomputer or network of workstations with one or more disks attached to each node. Data chunks are distributed across the disks attached to nodes to achieve I/O parallelism during query processing. A data chunk is assigned to only one disk, and is stored in a file on that disk. An *index*, constructed from the MBRs of the chunks, is used to find the chunks that intersect a query region during query processing.

Loading a dataset into ADR is accomplished in four steps: (1) partition the dataset into data chunks, (2) compute placement information, (3) create an index, and (4) move data chunks to the disks according to placement information, and register the dataset in ADR. The placement information describes how data chunks are distributed across the disks in the system. After data chunks are stored in the disk farm, dataset catalogs are updated to register the new dataset in ADR. To load a dataset into ADR, the user partitions the dataset into chunks and stores the chunks in a set of files. ADR provides utility programs to compute the placement, create an index (i.e. an R-tree index), and move data chunks from user-created files to the disks in the system. The user is required to assist the utility programs by providing a set of meta-data files that describe the data file location and meta-data (e.g., minimum bounding rectangle) for the chunks within the data files. Optionally, the user can move the data files to disks manually, if the chunks are already declustered across the data files, and use a user-defined index. To register a dataset, ADR provides utility programs that will take a file that describes meta-data for the dataset such as the user-defined name and a short description of the dataset and index, and will assign an id to the dataset and update the ADR dataset catalogs.

2.2 Client

A client process is a program, sequential or parallel, which is implemented for a specific application. It generates requests to ADR, and post-processes the output returned by the back-end. For example, a client process could be a graphical interface for end users to both generate requests and display the output returned by the back-end.

2.3 Front-end

The front-end consists of one or more *application front-ends* and a single *ADR front-end*. The ADR front-end and application front-ends are separate processes, potentially running on different

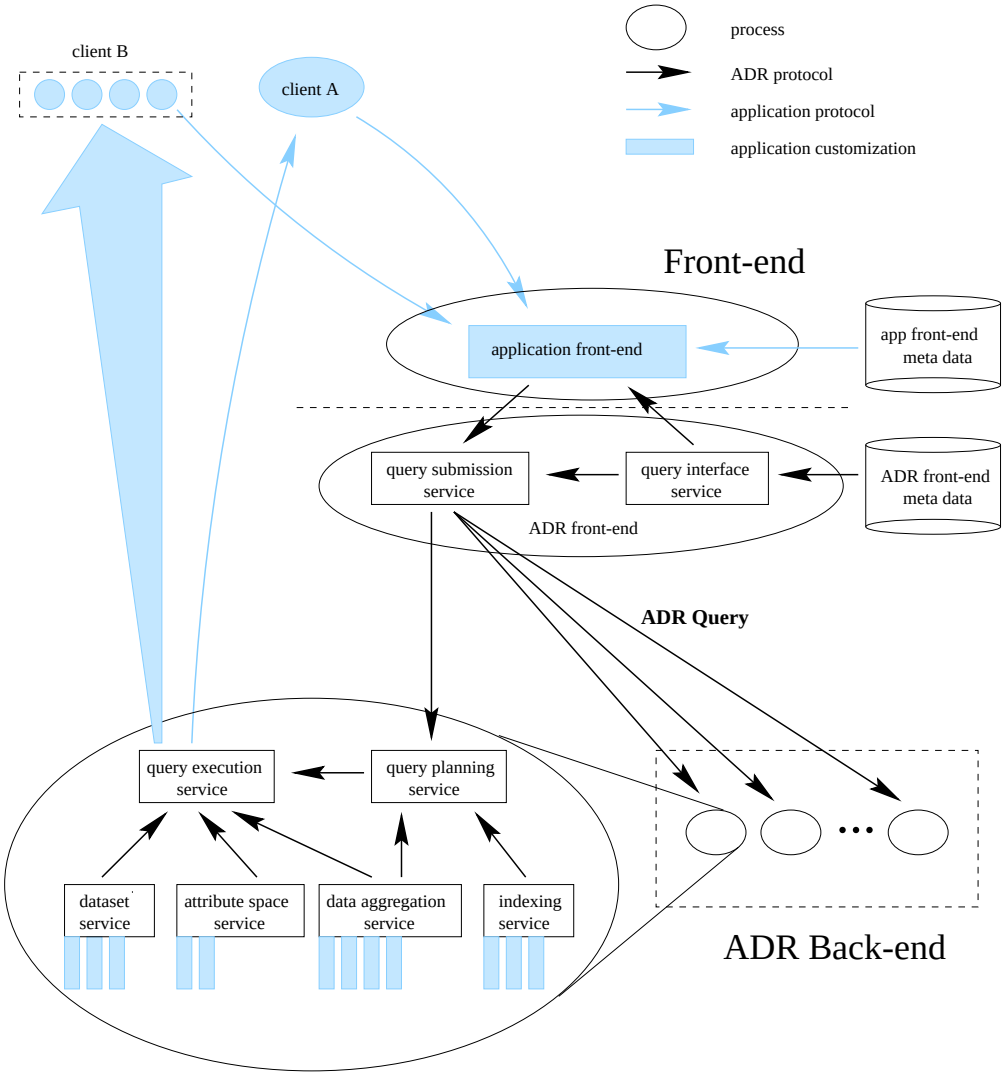

Figure 2: A complete application suite implemented as a customized ADR application. The shaded portion represents either application-specific programs or the part of the ADR services customized for the application. The arrows represent interaction between different entities: the darker ones use the protocol or interface defined by ADR and the lighter ones follow the protocol defined by the application. The shaded bars attached to the data service, the attribute space service, the data aggregation service and the indexing service represent functions added to ADR by the application developer as part of the customization process. Also, client A is shown as a sequential program while client B is shown as a parallel program.

machines.

An application front-end, implemented by the application developer, is responsible for receiving application client requests and translating each client request into *ADR queries*. The existence of the application front-end allows clients to communicate with the front-end using an application-specific protocol, which can be tailored to be efficient and natural for that application. It also allows additional application-specific meta-data to be stored with the front-end so that simple client requests that involve only the application meta-data (but not the actual datasets stored at the back-end) can be answered directly by the application front-end. The *query interface service* of ADR provides a set of utility classes that can be used in the application front-end to connect to the ADR front-end, to query ADR to get information about datasets and user-defined functions registered in ADR, and to create and submit ADR queries.

The ADR front-end interacts with application clients/front-ends and the ADR back-end. Since clients can connect and generate requests in an asynchronous manner, the existence of a front-end relieves the back-end from being interrupted by clients during query processing. The ADR front-end uses the *query interface service* to interact with application front-ends, and the *query submission service* to schedule multiple queries received from one or more application front-ends and to submit them to the back-end. Upon receiving a query, the query submission service places it into a query pool, to be forwarded to the back-end. When the back-end signals that it is ready, a scheduling policy determines which queries from the query pool are sent to the back-end. All queries sent to the back-end as a group are processed simultaneously. Currently, the query submission service uses a simple first-in-first-out policy, and can be instructed at start-up time to either send all the queries in the query pool to the back-end when requested, or always send a fixed number of queries from the head of the queue.

An **ADR Query** consists of:

- references to one or more input datasets, and for each dataset,
  - a range query, defined in the underlying multi-dimensional attribute space of the dataset,
  - a reference to an index, for finding data elements that intersect the range query,
  - a reference to a dataset iterator function, to access individual data elements,
  - a reference to a projection function, to project input data points in the input attribute space to data points in the output attribute space.
- a reference to an accumulator data structure, to hold the intermediate results,
- a reference to an aggregation function, to aggregate input elements with output/accumulator elements,
- a specification of how to handle the output (e.g., send the data to the client via sockets or store the data in files).

A reference to a dataset or to a user-defined function is the id of the dataset or function, which is assigned when the dataset or the function is registered in ADR. ADR provides utility classes, which the application front-end can use, to retrieve the id information.

2.4 Back-end

The back-end is responsible for storing the datasets and carrying out the required data retrieval and application-specific data processing for queries. As is shown in Figure 2, the back-end consists

of six services, which operate in each back-end process. Two of these services are internal services used by back-end processes to schedule and process the queries:

- *The query planning service*, which determines a query plan that can be used to efficiently process a set of queries based on resource availability.
- *The query execution service*, which manages all the resources in the system and carries out the query plans generated by the query planning service.

The remaining four services are customizable services: *attribute space service*, *dataset service*, *indexing service*, and *data aggregation service*. We briefly describe these services in the following sections.

2.4.1 Attribute space service

The attribute space service manages the registration and use of multi-dimensional attribute spaces and projection functions. Currently, an attribute space is specified by the number of dimensions. Projection functions are used to project points between attribute spaces. They are specified by the domain and range attribute spaces and an algorithm for the mapping between them.

2.4.2 Dataset service

As was stated previously, ADR expects each of the datasets to be partitioned into data chunks, each chunk consisting of one or more data items from the same dataset. In the current version, the user is responsible for partitioning a dataset into data chunks. There is no restriction on the size and the structure of a chunk. The user may choose any size, possibly optimized for a particular dataset– each chunk in the same dataset may have a different size. In addition, the user determines how the data items in a chunk are organized, which is possibly optimized or natural for a particular dataset. Therefore, a chunk in one dataset may have a different structure than a chunk in another dataset– only restriction is that all chunks in a dataset should have the same structure. The dataset service manages user-defined iterator functions that understand the structure of a chunk in a specific dataset. An iterator function is used to access individual data items in a data chunk.

2.4.3 Indexing service

The indexing service manages various indices for the datasets stored in the ADR back-end. The key function of an index is that, given a multi-dimensional range query in its underlying attribute space, it returns the disk locations for the set of data chunks that may contain data items that fall inside the given range query. To create an index, the indexing service uses the *MBR* for each chunk in the dataset and the physical location of each chunk on disk. ADR allows the application developer to optionally specify an indexing algorithm. This can be done by defining an index object derived from the ADR index class and implementing, among other virtual functions, the search function. By default, ADR uses a variant of an R-tree [3] index.

2.4.4 Data aggregation service

The data aggregation service manages the user-provided functions to be used in aggregation operations, and also encapsulates the data types of both the intermediate results used by these functions and the final outputs generated by these functions. An *accumulator* is used during query processing

to hold partial results generated by the aggregation functions. As for input datasets, an accumulator has an underlying attribute space, and each of its elements is associated with a point in the attribute space. An accumulator data type is defined by a user-defined class derived from the ADR accumulator base class. An accumulator data type implements virtual functions to allocate the accumulator elements under a given memory constraint (imposed by ADR) and to access individual accumulator elements that fall inside a given region in its underlying attribute space.

An output data type defines the data structure of the final output generated from processing an ADR query. Its major tasks are to hold the final results at the end of the query and to define the order in which values are communicated back to the requesting client, which must able to correctly interpret what it receives. An output data type is defined as a user-defined class derived from the ADR output base class.

An aggregation function is encapsulated as a user-defined class derived from the ADR aggregation base class. It implements virtual functions to initialize accumulator elements before aggregation takes place, merge the values of an input data item with an accumulator element, merge the values of a set of accumulator elements with another matching set of accumulator elements, and post-process the accumulator into the desired final output after all data aggregation has completed. Functions to merge corresponding accumulator elements are needed because the query execution service allows each back-end process to compute partial results into a local accumulator, which are then merged across back-end processes.

2.4.5 Customizing ADR Back-end

To build a version of ADR customized for a particular application, the application developer has to provide functions:

- to search (via an index) for data chunks that intersect the range query,
- to access the individual input data items in a retrieved chunk,
- to create and manage accumulator data structures,
- to project points between the input and output attribute spaces,
- to aggregate input data items that project to the same output item (by aggregating with the corresponding accumulator element),
- to create the final output from accumulator elements.

Customizing the ADR back-end for a specific application involves customizing the attribute space service, the dataset service, the indexing service, and the data aggregation service, as shown by the shaded bars attached to those services in Figure 2. Customization in ADR is currently achieved through C++ class inheritance. That is, for each of these four services, ADR provides a set of C++ base classes with virtual functions that must be implemented by derived classes. Adding an application-specific entry into a modular service requires the application developer to define a class derived from an ADR base class for that service and provide the appropriate implementations of the virtual functions. For example, the indexing service manages indices that allow ADR to efficiently locate the data items of datasets specified by a range query in persistent storage. It includes an index base class that contains a search function, among other virtual functions, that is expected to implement this functionality. Adding a new index therefore requires the definition of a new index class derived from the ADR base class and the proper implementation of the search function.

In addition to defining derived classes from an ADR base classes and providing the appropriate implementation of the virtual functions, users also need to provide *constructor functions* for many of the derived classes. A constructor function for a derived class is a regular C++ function (as opposed to a class member function) that when invoked, will dynamically create an instance of the derived class. This is necessary since ADR cannot dynamically create an instance of a derived class simply through its base class. In most cases, a constructor function simply creates an instance of the derived class with the C++ `new` operator and returns the pointer. Note that the C++ `new` operator would initialize the instance using the constructor member function of the derived class, as is defined in the C++ language.

ADR provides utility programs to register user-defined constructor functions. The registration process allows ADR to setup function pointers to registered constructor functions, so that the appropriate constructor function can be called at run-time when needed. Section 6.4 describes the process of constructor function registration in more details. Not all derived classes require constructor functions. In Section 4, C++ function signatures for constructor functions are listed along with the ADR base class definitions when constructor functions are required for the derived classes.

2.5 Query Processing in ADR

As was stated in Section 2.4, the processing of a query is carried out by *query planning* and *query execution* services.

2.5.1 Query planning service

To be able to efficiently integrate data retrieval and processing on a parallel machine, ADR manages the allocation and scheduling of all resources, including processor, memory, disk bandwidth and network bandwidth. The task of the query planning service is to determine a schedule for the use of the available resources to satisfy a set of queries. Given the nature of the computations supported by ADR, use of several of these resources is not independent (e.g., it is not possible to use disk bandwidth without having memory to store the data being transferred from disk). Note that changing the order input data items are retrieved cannot affect he correctness of the result, which is one of the requirements for an ADR application. Therefore, the associative and commutative nature of the aggregation operations must be leveraged to form loosely synchronized schedules – the schedules for individual processors need not proceed in lock-step and only need to synchronize infrequently. The ADR query planning service creates schedules based on requirements for memory, processor and network bandwidth. A query plan specifies how parts of the final output are computed and the order the input data chunks are retrieved for processing.

Planning is carried out in two steps; *tiling* and *workload partitioning*. In the tiling step, if the accumulator is too large to fit entirely into the memory, it is partitioned into *tiles*. Each tile contains a distinct subset of the accumulator elements, so that the total size of a tile is less than the amount of memory available for the accumulator. Tiling of the accumulator implicitly results in a tiling of the input dataset. Each input tile contains the input chunks that map to the corresponding accumulator tile. During query processing, each accumulator tile is cached in main memory, and input chunks from the corresponding input tile are retrieved. In the workload partitioning step, the workload associated with each tile (i.e. aggregation of data items in input and accumulator chunks) is partitioned across processors. This is accomplished by assigning each processor the responsibility for processing a subset of the input and/or accumulator chunks. Currently, ADR implements a workload partitioning strategy referred to as *replicated accumulator*. In this strategy, each back-end

node is assigned the responsibility to carry out processing associated with its local input chunks. The accumulator elements in a tile are effectively replicated across all nodes.

For each query that an ADR back-end process receives, the *query planning service* uses the selected index from the *indexing service* to locate the set of data items that need to be retrieved for the given query. Then, it uses the *data aggregation service*, in conjunction with knowledge of the amount of resources available, to generate a *query plan*. The query planning service uses the user-defined functions implemented in data aggregation service to partition the user-defined accumulator into tiles. The output of the planning service consists of a set of ordered lists of *input chunks*, one list per disk in the machine configuration for each accumulator tile.

2.5.2 The query execution service

Execution of a query in ADR is done through the *query execution service*, which manages all the resources in the system and carries out the query plan generated by the query planning service. The query execution service receives the query plan and carries out the data retrieval and processing. The execution service iterates through accumulator tiles, and processes the query one accumulator tile at a time. That is an accumulator tile is allocated space in each back-end node and initialized using user-defined methods in the data aggregation service. During the processing, the query execution service in each back-end node uses (1) the *dataset service* to navigate through the input data items in each data chunk retrieved from the local disks, (2) the selected projection function from the *attribute space service* to map input data items to output items, (3) the selected aggregation function from the data aggregation service to combine the values of the input data items that project to the same output item, with the corresponding accumulator item. Finally, the corresponding portion of the output is created from the accumulator tile, using the user-defined output methods, and is returned to the client as specified by the query. Therefore, the processing of a query on a back-end processor progresses through the following phases for each tile:

1. **Initialization**. Accumulator elements in the current tile are allocated space in memory and initialized.

2. **Local Reduction**. Corresponding input data chunks on the local disks of each back-end processor are retrieved and input elements are aggregated into the accumulator elements allocated in each processor's memory in phase 1.

3. **Global Combine**. Partial results computed in each processor in phase 2 are combined across all processors to compute final results.

4. **Output Handling**. The final output for the current tile is computed from the corresponding accumulator elements computed in phase 3.

A query iterates through these phases repeatedly until all tiles have been processed and the entire output dataset is handled. When multiple queries are processed simultaneously by the ADR back-end, each query independently progresses through the four query execution phases.

To reduce query execution time, ADR overlaps disk operations, network operations and processing as much as possible during query processing. Overlap is achieved by maintaining explicit queues for each kind of operation (data retrieval, message sends and receives, data processing) and switching between queued operations as required. Pending asynchronous I/O and communication operations in the operation queues are polled and, upon their completion, new asynchronous operations are initiated when more work is expected and memory buffer space is available. Data chunks are therefore retrieved and processed in a pipelined fashion.

3 Customizing ADR: Front-end

ADR has a set of utility classes, which provide methods to connect to the ADR front-end, to inquire into ADR to get information about datasets and user-defined methods, and to create and submit ADR queries.

We first briefly describe some of the ADR classes and types that are passed as parameters to or returned from the methods in ADR utility classes for front-end, which are described in the following sections. More detailed descriptions of these classes can be found in Appendix A.

- A set of data types for various kinds of identification numbers, currently all are of type `unsigned int`:
  - `T2_FrontEndError`: Error no returned from a call to methods to connect to ADR front-end.
  - `T2_DSID`: Dataset id
  - `T2_IteratorID`: Iterator id
  - `T2_IndexID`: Index id
- `T2_Box`: A hyper-box in some multi-dimensional attribute space (*e.g.*, (10,20,50)-(40,100,60)); it consists of two points, a `low` point and a `high` point, which specify the lower and upper bounds of the hyper-rectangle, respectively.
- `T2_UsrArg`: A class used to represent application-dependent arguments, stored as an array of bytes. The user code would actually use either one of the following classes derived from class `T2_UsrArg`:
  - `T2_UsrArgWriter`: A class that allows data to be written into its buffer.
  - `T2_UsrArgReader`: A class that allows data to be read from its buffer.
- `T2_FEDatasetInquiryResults` This class encapsulates the results returned from a call to inquiry functions for dataset information.
- `T2_FEFunctionInquiryResults` This class encapsulates the results returned from a call to inquiry functions for information about user-defined methods.

3.1 Connecting to ADR Front-end

The **T2_FrontEnd** class provides support to connect to the ADR front-end. It also provides methods to query ADR and create and submit ADR queries. The definition of `T2_FrontEnd` is shown in Figure 3:

- `connectT2FrontEndByHostname` is used to connect to the ADR front-end using the hostname of the machine ADR front-end is running on. It returns `true` on success, `false` on error, and sets the error value to the error no returned by the socket library.
  - `hostname` is the name of the ADR front-end host.
  - `port` is the port number, which ADR front-end is listening to for client connections.
- `connectT2FrontEndByAddress` is used to connect to the ADR front-end using the IP address of the host ADR front-end is running on. It returns `true` on success, `false` on error, and sets the error value to error no returned by the socket library.

  - `ipaddr` is the IP address of the ADR front-end host.
  - `port` is the port id, which ADR front-end is listening to for connections.

- `disconnectT2FrontEnd` closes connection to the ADR front-end.
- `getErrorVal` returns the error no.
- `errorValToString` converts error no to a short description of the error.
- `getNumberBackEndNodes` returns the number of ADR back-end processes.
- `getT2FrontEndEndianess` returns the endian representation of ADR front-end host machine. It returns
  - `T2_BigEndian` if ADR front-end is running on a big-endian machine,
  - `T2_LittleEndian` if ADR front-end is running on a little-endian machine,
  - `T2_UnknownEndian` if the endian representation of the machine cannot be determined.
- `getMyEndianess` returns the endian representation of the machine which the calling process is running on.
- `submitQBatch` is used to submit a query batch, which consists of one or more ADR queries, to the ADR front-end. It returns `true` on success (i.e. if query batch is submitted successfully), `false` otherwise.
  - `qbatch` is the query batch to be submitted to ADR front-end (see Section 3.3 for a description of `T2_QBatch`.)

3.2 Querying ADR Front-end

The **T2_FrontEnd** class also provides methods to query ADR front-end for information about datasets and user-defined methods:

- `inquireDatasetExactMatch` is used to retrieve information about datasets. It takes the name of the dataset as an argument, and retrieves dataset information for the dataset, whose name matches the argument. It returns `true` if no error occurs, `false` otherwise.
  - `dataset_name` The name of the dataset for which the information is to be retrieved.
  - `fields` encodes which fields of dataset information should be returned as a result of the inquiry. The meta-data for datasets contains the following field identifiers, which can be set in the `fields` parameter:
    * `dataset_datasetid_field` should be set to retrieve the id of the dataset as one of the fields of the result of the inquiry.
    * `dataset_datasetname_field` should be set to retrieve the name of the dataset.
    * `dataset_datasetdescription_field` should be set to retrieve a short description of the dataset.
    * `dataset_blobobj_field` should be set to retrieve the user-defined binary meta-data object. User-defined binary meta-data object contains application specific information that can be processed by the application client/front-end. For example, it may contain a thumbnail image for an image processing application.

```
#include "t2_frontend.h"

class T2_FrontEnd {
public:
  // T2_FrontEnd constructor
  T2_FrontEnd();

  // connect to/disconnect from ADR front-end process
  // return true if succeed, false otherwise;
  // when return false, sets errno to the errval returned by socket library
  bool connectT2FrontEndByHostname(const char* hostname, short port);
  bool connectT2FrontEndByAddress(const char* ipaddr, short port);
  void disconnectT2FrontEnd();

  // get error value
  T2_FrontEndError getErrorVal() const;
  // convert error value to string, usually for printout
  static const char* errorValToString(T2_FrontEndError e);
  const char* errorValToString() const;

  // number of back-end nodes
  u_int getNumberBackEndNodes() const;

  // get endianess of the ADR front-end machine and the local machine
  int getT2FrontEndEndianess() const;
  int getMyEndianess() const;

  // submit a query batch
  bool submitQBatch(T2_QBatch& qbatch);

  // inquiry methods for datasets and functions for exact matching
  // or regular expression matching;
  bool inquireDatasetExactMatch(const char* dataset_name, const int fields,
                                const size_t max_totblobsize,
                                T2_FEDatasetInquiryResults& results);
  bool inquireDatasetRegExp(const char* pattern, const int fields,
                            const size_t max_totblobsize,
                            T2_FEDatasetInquiryResults& results);
  bool inquireFunctionExactMatch(const char* function_name,
                                 const T2_UDFType ftype, const int fields,
                                 T2_FEFunctionInquiryResults& results);
  bool inquireFunctionRegExp(const char* pattern, const T2_UDFType ftype,
                             const int fields,
                             T2_FEFunctionInquiryResults& results);
};
```

Figure 3: The ADR utility class to connect to ADR front-end, inquire into ADR for information about datasets, user-defined methods, and submit queries.

* `dataset_blobsize_field` should be set to retrieve the size of the user-defined binary meta-data object.
* `dataset_iteratorname_field` should be set to retrieve list of the names of the iterators for the dataset. Note that iterator ids are local to the dataset and the id of an iterator is its position in the list.
* `dataset_indexname_field` should be set to retrieve the name of the dataset index.
* `dataset_indexid_field` should be set to retrieve the id of the index. The index ids are global and unique, because multiple datasets may use the same index method.

Setting multiple fields in `fields` parameter is achieved by a bitwise-OR of field identifiers. For example, if `fields` is set as follows

```
fields = dataset_datasetid_field | dataset_indexid_field
```

Then, only the dataset id and the index id are returned as the result of the inquiry.

- `max_totblobsize` limits the size of user-defined binary meta-data object returned in the result. If the size of the binary object stored in ADR is larger than `max_totblobsize`, the binary object is truncated.
- `results` holds the results of the query. Note that if there are multiple datasets that satisfy the query, then `results` parameter holds an array of results, each entry of which is the query result for each of the datasets.

- `inquireDatasetRegExp` is used to retrieve information about datasets. The main difference of this method from `inquireDatasetExactMatch` is that it takes a regular expression (`pattern`) instead of the name of the dataset. It returns `true` if no error occurs, `false` otherwise.
- `inquireFunctionExactMatch` is used to retrieve information about user-defined constructor functions (see Section 4) for various services.
  - `function_name` is the name of the constructor function, for which the information is to be retrieved.
  - `ftype` is the type of the function. The valid values are
    * `T2_UDF_Unknown` to retrieve information about all constructor functions,
    * `T2_UDF_AccMeta` to retrieve information about constructor function for accumulator meta-data object,
    * `T2_UDF_Aggregation` to retrieve information about constructor function for aggregation object,
    * `T2_UDF_Projection` to retrieve information about constructor function for projection object.
  - `fields` encodes which fields of function information should be returned as a result of the inquiry. The meta-data for constructor functions contains the following field identifiers that can be set in the `fields` parameter:
    * `function_id_field` should be set to retrieve the id of the function.
    * `function_name_field` should be set to retrieve the name of the function.
    * `function_description_field` should be set to retrieve a short description of the function.

Setting multiple fields in `fields` parameter is achieved by a bitwise-OR of field identifiers. For example, if `fields` is set as follows

```
fields = function_id_field | function_name_field
```

Then, only the id and the name of the constructor function are returned as the result of the query.

- `results` holds the results of the inquiry. Note that if there are more than one function that satisfy the query, then `results` parameter holds an array of results, each entry of which is the query result for each of the functions.

- `inquireFunctionRegExp` is used to retrieve information about user-defined constructor functions (see Section 4) for various services.
  The main difference of this method from `inquireFunctionExactMatch` is that it takes a regular expression as a parameter (`pattern`), instead of the name of the function. It returns `true` on success, `false` otherwise.

3.3 Formulating an ADR Query

ADR provides utility classes that contain methods to create ADR queries and query batches. An ADR query consists of:

- a reference to an accumulator data structure,
- a reference to aggregation function,
- references to one or more input datasets, and for each dataset,
  - a reference to dataset iterator function,
  - a reference to index,
  - a reference to projection function,
  - a range query, defined in the underlying multi-dimensional attribute space of the dataset.
- a specification of how to handle the output (e.g., send the data to the client via sockets).

The **T2_QSpec** class provides methods to create an ADR query. Its definition is shown in Figure 4:

- `getAccID` is used to assign/retrieve the accumulator meta-data object id.
- `getAccConstructorArg` is used to specify user defined argument for the constructor function for the accumulator meta-data object. The definition of `T2_UsrArg` is given in Section A.
- `getAggrID` is used to assign/retrieve the aggregation function id.
- `getAggrConstructorArg` is used to specify user defined argument for the constructor function for the aggregation object.
- `setNumberDatasets` is used to specify the number of input datasets accessed in the query.
- `getNumberDatasets` returns the number of datasets specified in the query.

```
#include "t2_frontend.h"

class T2_QSpec {
public:
  T2_QSpec(const u_int num_input_datasets = 0);

  u_int& getAccID();
  T2_UsrArg& getAccConstructorArg();
  u_int& getAggrID();
  T2_UsrArg& getAggrConstructorArg();

  void setNumberDatasets(u_int num_input_datasets);
  u_int getNumberDatasets() const;
  T2_QSpecDataset& getDatasetSpec(u_int dataset_id);

  T2_QSpecOutput& getOutputSpec();
};
```

Figure 4: The ADR utility class to create a query.

- `getDatasetSpec` is used to specify query parameters for the dataset, the id of which is given in `dataset_id`. See Figure 5 for the definition of `T2_QSpecDataset` class.
- `getOutputSpec` is used to specify how the output should be handled. See Figure 6 for the definition of `T2_QSpecOutput` class.

The **T2_QSpecDataset** class provides methods to specify query parameters for datasets accessed in the ADR query. The definition of this class is shown in Figure 5:

- `getDatasetID` is used to specify the id of the dataset.
- `getIteratorID` is used to specify the id of the dataset iterator.
- `getIndexID` is used to specify the id of the index.
- `getProjID` is used to specify the id of the projection method.
- `getIndexConstructorArg` is used to specify the user-defined argument to the constructor function for the index object.
- `getProjConstructorArg` is used to specify the user-defined argument to the constructor function for the projection function object.
- `getQueryBox` is used to specify the range query. The range query is a multi-dimensional box of type `T2_Box` (see Appendix A for the definition of `T2_Box`).

The **T2_QSpecOutput** provides methods to specify how the output should be handled. Its definition is shown in Figure 6:

- `setOutputHandleType` is used to set the output handling type. The valid values for the input parameter `type` are:

```
#include "t2_frontend.h"

class T2_QSpecDataset {
public:
  T2_QSpecDataset();

  T2_DSID& getDatasetID();
  T2_IteratorID& getIteratorID();
  T2_IndexID& getIndexID();
  T2_UsrArg& getIndexConstructorArg();
  u_int& getProjID();
  T2_UsrArg& getProjConstructorArg();
  T2_Box& getQueryBox();
};
```

Figure 5: The ADR utility class to specify query parameters for a dataset in an ADR query.

  - `t2_oBFile` to write the output into a binary file.
  - `t2_oSocket` to send the result from back-end nodes to the application client via UNIX sockets.
  - `t2_oMC` to send the results from back-end nodes to the application client via Meta-Chaos library. This option can be useful when the client is a parallel program. Meta-Chaos library provides functions to exchange distributed data structures between two parallel programs. **The Meta-Chaos interface for ADR will be included in this document in the future**.

- `enableT2Protocol` is used to enable ADR protocol when sending the output to the client through sockets.
- `disableT2Protocol` is used to disable ADR protocol so that application client can use only the application specific header information.
- `useT2Protocol` returns `true` if ADR protocol is enabled. It returns `false` otherwise.
- `setHostName` is used to specify the name of the machine the client code is running on. If output handling type is specified as `t2_oSocket`, then ADR back-end nodes use that name to open socket connections to send output data.
- `setPortNumber` is used to specify the port, which the client is listening for connections from the back-end nodes.
- `setFilePrefix` is used to specify the prefix for files to be created when the output is written to binary files. Each processor creates a file for each tile of the accumulator data structure. The prefix value is augmented by processor no, and the tile no.
- `setClient` is used to specify the user-defined name of the client when Meta-Chaos library is used for sending the data. The client name is used by Meta-Chaos libraries to initiate connections with the client program. The client should also use Meta-Chaos library calls to accept Meta-Chaos connections from ADR back-end nodes.

- `setNumberClients` is used to specify the number of processes running in the client program.
- `setOffBandTag` is used to specify the message tag for off-band messages to be exchanged between ADR back-end and the client.
- `setDataTag` is used to specify the message tag for sending data values from ADR back-end to the client using Meta-Chaos library.

ADR allows multiple queries to be submitted in a single query batch. The queries in a batch are processed concurrently in the ADR back-end. The **T2_QBatch** provides methods to create query batches. Its definition is shown in Figure 7:

- `setNumberQueries` sets the number of queries in the query batch to the value specified in `num_of_queries`. It effectively resizes the query batch and throws away all the existing queries in the current batch.
- `getNumberQueries` returns the number of queries in a query batch.
- `getQuerySpec` returns a reference to the query object specified by `query_id` in the query batch. The referenced query object can be used to create an ADR query using the query specification classes and methods presented in the previous sections.

A query batch is submitted from the application client/front-end to the ADR front-end using `submitQBatch` method of `T2_FrontEnd` class (see Section 3.1).

```
#include "t2_frontend.h"

enum T2_OutputHandleType {
     t2_oUnknown = 0, // unknown
     t2_oBFile,       // output to a binary file
     t2_oSocket,      // send to client over sockets
     t2_oMC           // send to client using Meta-Chaos library
};

class T2_QSpecOutput {
public:
  T2_QSpecOutput();

  // accessing output handle type
  T2_OutputHandleType setOutputHandleType(const T2_OutputHandleType& type);

  // enabling/disabling T2 protocol (default is enabled)
  void enableT2Protocol();
  void disableT2Protocol();
  bool useT2Protocol() const;

  // access methods for outputing to a socket
  const char* setHostName(const char* hostname);
  int setPortNumber(int port_id);

  // access methods for outputing to a binary file
  const char* setFilePrefix(const char* file_prefix);

  // access methods for outputing through meta-chaos
  const char* setClient(const char* client);
  u_int setNumberClients(u_int num_of_clients);
  int setOffBandTag(int off_tag);
  int setDataTag(int data_tag);
};
```

Figure 6: The ADR utility class to create specification of the output in an ADR query.

```
#include "t2_frontend.h"

class T2_QBatch {
public:
  T2_QBatch(u_int num_of_queries = 0);

  // access methods
  u_int getNumberQueries() const;
  T2_QSpec& getQuerySpec(u_int query_id);

  // resize the batch and throw away all the existing query specs
  void setNumberQueries(u_int num_of_queries);
};
```

Figure 7: The ADR utility class to create a query batch.

4 Customizing ADR: Backend Services

In this section we describe the interface for customizing ADR backend services. The current implementation of ADR services is based on C++ classes. Customization of these services relies on C++ class inheritance and implementation of virtual functions. That is user defined indexes, accumulator objects, projection and aggregation operations are implemented as classes and methods derived from appropriate base classes provided by ADR. Users are also required to implement a *constructor function* for each derived class because a user-defined object cannot be instantiated by invoking the constructor of its base class. A constructor function receives arguments stored in the ADR query, and is expected to create an object of the user-defined class. The user-defined constructor functions for classes derived from the same ADR base class are all stored in a *constructor function list*, accessed by ADR during query processing. The location of a constructor function in the list defines the id of the corresponding user-defined derived class. An ADR query specifies the ids of user-defined objects to be used during query processing. Note that ADR infrastructure should be recompiled every time a new user-defined class is added.

In the following sections we describe the definitions of base classes and their virtual methods for implementing customized ADR services.

4.1 Overview of ADR Utility Classes

In this section we briefly describe some of the ADR utility classes that are passed as parameters to the methods in ADR base classes. More detailed descriptions of these utility classes can be found in Appendix A.

- A set of data types for various kinds of identification numbers, currently all are of type `unsigned int`:
  - `T2_ProcID`: Processor id
  - `T2_DSID`: Dataset id
  - `T2_IteratorID`: Iterator id
  - `T2_IndexID`: Index id
  - `T2_ChunkID`: Chunk id used in ADR indexing service
  - `T2_UDFRet`: Error value returned from a user-defined method in a derived class. Currently, there are only two values; `T2_UDFRet_OK` indicates that the method returned successfully, and `T2_UDFRet_ERROR` indicates that an error occured.
- `T2_System`: An object that provides system information such as the number of processors and the local processor id.
- `T2_Point`: A point in some multi-dimensional attribute space (*e.g.*, (3, 50, 29) in a 3-dimensional space); each coordinate of the point is a stored as a `float`.
- `T2_Box`: A hyper-box in some multi-dimensional attribute space (*e.g.*, (10,20,50)-(40,100,60)); it consists of two points, a `low` point and a `high` point, which specify the lower and upper bounds of the hyper-rectangle, respectively.
- `T2_Region`: an unordered list of `T2_Box`'es in some multi-dimensional attribute space (*e.g.*, {(1,5)-(2,7), (10,4)-(11,5)} represents the following points: (1,5), (1,6), (1,7), (2,5), (2,6), (2,7), (10,4), (10,5), (11,4), (11,5))

- `T2_Cluster`: A set of `(void*, size_t)` pairs, each of which represents a pointer to a contiguous data block and its size.
- `T2_UsrArg`: A class used to represent application-dependent arguments, stored as an array of bytes. This class is usually used in user-defined constructor functions of derived classes; the callee that receives an object of this class is responsible for parsing the arguments correctly. The user code would actually see either one of the following classes associated with class `T2_UsrArg`:
  - `T2_UsrArgWriter`: A class that allows data to be written into its buffer.
  - `T2_UsrArgReader`: A class that allows data to be read from its buffer.
- `T2_Array<`*Type*`>`: A template for an array of objects of class *Type*.
- `T2_VArray<`*Type*`>`: A template for a variable-size array of objects of class *Type*. The size of the array expands as new objects of class *Type* are appended.
- `T2_BlockRequest`: An object to store a `file id`, an `offset` and a `size` for a data chunk.
- `T2_AccMsgBuffer`: A buffer for communicating the accumulator elements during the global combine phase. It consists of a `void*` pointer and a `size`.
- `T2_OutputBuffer`: A buffer that the final output uses for flattening out its data into contiguous space when necessary. It consists of a `void*` pointer and a `size`.
- `T2_OutputDataPtrList`: An ordered list of pairs of `char*` pointers and data `sizes`.
- `T2_ClusterInfoList`: It contains information for a list of input data chunks of all datasets accessed by a given query. Each entry corresponds to a single chunk.
- `T2_ClusterAuxInfoWriter`: allows the user-defined meta-data for a chunk, retrieved during index search, to be written into a buffer.
- `T2_ClusterAuxInfoRead` is used to read the user-defined meta-data for a chunk from the buffer encapsulated by `T2_ClusterAuxInfoWriter`.

4.2 Indexing Service

An index is used to efficiently find all data chunks that intersect with a given query box. ADR provides a base class (**T2_Index**) for implementing user-defined index methods. The definition of the base class and the constructor function are shown in Figure 8:

- `fetchInit` This method initiates the search. ADR first calls this method, before subsequent calls to `fetch` method. This method may create a state for the current query that can be used in calls to `fetch` to retrieve meta-data information for data chunks that intersect with the query box. `fetchInit` method returns `T2_UDFRet_OK` on success or `T2_UDFRet_ERROR` if there is an error.
  - `system` This input parameter can be used to query system information such as number of processors and local processor id.
  - `query_box` This input parameter holds the hyper-box of the range query submitted by the application. The hyper-box is defined in the input attribute space.

```
#include "t2_index.h"

class T2_Index {
public:
  virtual T2_UDFRet fetchInit(const T2_System& system,
                              const T2_Box& query_box);
  virtual T2_UDFRet fetch(const T2_System& system, const T2_Box& query_box,
                          bool& end_of_data, T2_Box& chunk_mbr,
                          T2_ClusterAuxInfoWriter& chunk_meta,
                          T2_VArray<T2_BlockRequest>& chunk_info);
};

// Constructor function for T2_Index object
typedef T2_UDFRet
(T2_IndexConstructor)(const T2_System& system, T2_UsrArgReader& user_arg,
                      const T2_Array<int>& index_fid, T2_Index*& index_obj);
```

Figure 8: The ADR base class for indexing service.

- `fetch` This method retrieves the meta-data information about the data chunks that intersect with the query bounding box. The `system` and `query_box` arguments are also passed to `fetch` so that if no state is created in `fetchInit` (i.e., it is an empty function), values of `system` and `query_box` need not be saved in `fetchInit`. `fetch` method returns `T2_UDFRet_OK` on success or `T2_UDFRet_ERROR` if there is an error.
  - `end_of_data` This output parameter is used to mark the end of search operation. It should be set to `true` in `fetch` if there are more entries to be returned from the search. It should be set to `false` if no more entries return from a call to `fetch`.
  - `chunk_mbr` This output parameter holds the minimum bounding rectangle of the data chunk in the input attribute space. Note that each data item in a multi-dimensional dataset is associated with point in the corresponding multi-dimensional attribute space. A chunk composed of such data items therefore is associated with a minimum bounding rectangle in the input attribute space.
  - `chunk_meta` This output parameter holds additional user-defined meta-data information associated with a chunk.
  - `chunk_info` This output parameter holds a list of block requests (`T2_BlockRequest`) for a chunk. A block is a physically contiguous segment in a data file. A chunk may be composed of one or more such segments from one or more data files. A block request (see Appendix A for definition of `T2_BlockRequest`) has three fields, which correspond to the logical file id (as is assigned in the dataset catalogs created when dataset is loaded into ADR), an offset into the corresponding data file, and the size of the file block.

The user-defined object derived from `T2_Index` is instantiated in the constructor function implemented by the application developer. The signature of the constructor function **T2_IndexConstructor** is given in Figure 8. The constructor function returns `T2_UDFRet_OK` on success or `T2_UDFRet_ERROR` on error:

- `system` This input parameter can be used to query the system information such as number of nodes and local node id.

```
#include "t2_prj.h"

class T2_ProjectFuncObj {
public:
  virtual T2_UDFRet project(const T2_System& system,
                            const T2_Box& query_box,
                            const T2_Region& accum_rgn,
                            const T2_Point& input_coord,
                            bool& succeed, T2_Region& output_rgn);
  virtual u_int getNumberInputDimensions();
  virtual u_int getNumberOutputDimensions();
  virtual T2_UDFRet projectBox(const T2_System& system, const T2_Box& inbox,
                               bool& succeed, T2_Box& outbox);
};

// Constructor function for T2_ProjectFuncObj object
typedef T2_UDFRet
(T2_ProjectFuncConstructor)(const T2_System& system, T2_UsrArgReader& user_arg,
                            T2_ProjectFuncObj*& proj_obj);
```

Figure 9: The ADR base class for attribute space service.

- `user_arg` This input parameter holds the user-defined argument to the constructor function. The value of `user_arg` is passed from application to the constructor function via the ADR query.
- `index_fid` This input parameter is an array of physical file descriptors of the local index files. The index files are opened by ADR so that constructor function can access the contents of the files by UNIX **read** call.
- `index_obj` This output parameter should point to the instance of the user-defined index object instantiated in the constructor function.

4.3 Attribute Space Service

This service provides a base class for implementing user-defined projection operations. Attribute spaces of input and output datasets in an application are currently defined only by the number of dimensions of each attribute space. A projection function is used to project points in the input attribute space to points in the output attribute space.

A user-defined attribute space object and projection function are implemented by a class derived from the **T2_ProjectFuncObj** base class. Figure 9 shows the definition of the base class and its constructor function. The application developer is required to implement the following virtual methods:

- `project` This method computes the projection of a point in input attribute space to one or more points in the output attribute space. The projected points are defined by a *region* (of type `T2_Region`), which is a list of hyper-boxes, in the output attribute space. `project` returns `T2_UDFRet_OK` on success or `T2_UDFRet_ERROR` on error.

  - `system` This input parameter can be used to query system information such as the number of processors and local processor id.
  - `query_box` This input parameter holds the hyper-box of the range query submitted by the application. The hyper-box is defined in the input attribute space.
  - `accum_rgn` This input parameter is a list of hyper-boxes, defined in the output attribute space, for the current accumulator tile. An accumulator may be a sparse data structure, and its pieces may be sparsely located in the output attribute space. Each hyper-box in `accum_rgn` corresponds to the minimum bounding rectangle of each such accumulator piece in the current accumulator tile.
  - `input_coord` This input parameter holds the coordinates of input point to be projected. The coordinates are defined in the input attribute space.
  - `output_rgn` This output parameter is the list of boxes and points the input point (`input_coord`) projects to.
  - `succeed` This output parameter is set to `true` if the coordinates of the input point is inside `query_box` and some of output points it projects to fall inside `accum_rgn`. It is set to `false` if the input point is not inside `query_box` and/or all of the output points it projects to fall outside `accum_rgn`. In that case the value of `output_rgn` is undefined.

- `getNumberInputDimensions` This method returns the number of dimensions in the input attribute space.

- `getNumberOutputDimensions` This method returns the number of dimensions in the output attribute space.

- `projectBox` This method projects a hyper-box in the input attribute space to a hyper-box in the output attribute space. ADR provides a default implementation for this method using the `project` method. The default implementation uses the end points of the input box to compute the end points of the output box. The application developer is required to provide an implementation if the end points of the input box does not project to the end points of the output box.

  - `inbox` This input parameter holds the coordinates of the input box in the input attribute space.
  - `outbox` This output parameter holds the coordinates of the output box in the output attribute space.

The constructor function **T2_ProjectFuncConstructor** is used by ADR to create an instance of user-defined class object. Application developer should implement a constructor function conforming to the signature definition given in Figure 9. The constructor function should return `T2_UDFRet_OK` on success or `T2_UDFRet_ERROR` on error:

- `system` This input parameter can be used to query the system information such as the number of nodes and the local node id.

- `user_arg` This input parameter holds the user-defined argument to the constructor function. The value of `user_arg` is passed from application to the constructor function in the ADR query.

- `proj_obj` This output parameter points to the instance of the user-defined class instantiated in the constructor function.

```
#include "t2_dataset.h"

class T2_Dataset {
public:
  virtual T2_UDFRet genIterator(const T2_System& system,
                                T2_IteratorID iterator_id,
                                const T2_Cluster& chunk,
                                T2_ClusterAuxInfoReader& chunk_meta,
                                const T2_Box& chunk_mbr,
                                const T2_Box& query_box,
                                const T2_Region& accum_rgn,
                                T2_ProjectFuncObj& proj_obj,
                                T2_Iterator*& iter_obj);
};

// Base class for dataset iterators.
class T2_Iterator {
public:
  virtual T2_UDFRet getNextElement(const T2_System& system,
                                   const T2_Cluster& chunk,
                                   bool& end_of_data,
                                   const void*& input_data,
                                   T2_Point& input_coord);
};

// Constructor function for dataset object
typedef T2_UDFRet
(T2_DatasetConstructor)(const T2_System& system, const T2_Array<int>& data_fid,
                        const T2_Array<int>& meta_fid, T2_Dataset*& dataset_obj);
```

Figure 10: The ADR base classes for dataset service.

4.4 Dataset Service

The major customization task to be carried out by the application developer for the dataset service is to define one or more iterators for the dataset. An iterator is used to iterate through the data elements of a data chunk retrieved during query processing. The user-defined iterator should understand the structure of data chunks, and should output the value and the coordinates of each data element in the data chunk. ADR provides two base classes, **T2_Dataset** and **T2_Iterator**, for application developer to implement iterators for a dataset. The definitions of these classes are given in Figure 10.

The `T2_Dataset` class has to be customized to instantiate an iterator object (derived from `T2_Iterator`) for each input dataset accessed during query processing:

- `genIterator` This method instantiates a user-defined iterator object for a chunk. It is called every time a chunk is retrieved into the memory from the local disks and is ready to be processed. `genIterator` returns `T2_UDFRet_OK` on success, or `T2_UDFRet_ERROR` on error.
  - `system` This input parameter can be used to query system information such as number

of processors and local processor id.

- `iterator_id` This input parameter contains the id of the iterator to be instantiated. A dataset is allowed to have more than one iterator. The iterator id is a field of the ADR query submitted by the application client/front-end.
- `chunk` This input parameter contains the data chunk retrieved from a data file stored on a local disk. Note that ADR uses customized indexing service (see Section 4.2) to generate the list of data chunks to be retrieved for processing the query.
- `chunk_meta` This input parameter holds additional user-defined meta-data information associated with a chunk. This meta-data is retrieved in indexing service (see Section 4.2).
- `chunk_mbr` This input parameter holds the minimum bounding rectangle of the retrieved data chunk.
- `query_box` This input parameter holds the hyper-box of the range query submitted by the application. The hyper-box is defined in the input attribute space. The query box is created by the application client/front-end to define the range query and is passed to dataset service via the ADR query.
- `accum_rgn` This input parameter is a list of hyper-boxes, defined in output attribute space, for the current accumulator tile. An accumulator may be a sparse data structure, and its pieces may be sparsely located in the output attribute space. Each hyper-box in `accum_rgn` corresponds to the minimum bounding rectangle of each such accumulator piece in the current accumulator tile.
- `proj_obj` This input parameter is a reference to the customized projection object instantiated by the indexing service (see Section 4.2).
- `iter_obj` This output parameter points to the user-defined iterator object instantiated by a call to `genIterator` method.

The `proj_obj`, `accum_rgn`, and `query_box` parameters can be used to clip a data chunk. Using these parameters, the iterator can quickly throw away the set of data elements that do not intersect with the query box and the accumulator bounding box, and therefore that need not be processed. The application developer has to implement a derived class of `T2_Iterator` base class for a user-defined iterator:

- `getNextElement` This method is used to iterate through the elements of a data chunk. It outputs the data element, and its coordinates in the input attribute space. This method is called by ADR until no more elements are output. `getNextElement` returns `T2_UDFRet_OK` on success, or `T2_UDFRet_ERROR` on error.
  - `end_of_data` This output parameter is used to mark the end of iteration. It should be set to `true` in the method if there are data elements to be output, and should be set to `false` otherwise.
  - `input_data` This output parameter is a pointer to the input data element extracted from the data chunk. The memory space to store the data element may be allocated in the method. However, if the data elements are regularly laid out in the data chunk, the `input_data` can simply point to a location in the data chunk. Thus, to prevent a memory deallocation error, ADR does not delete the object pointed by a pointer returned from a user-defined method. As a result, if memory space is allocated for a data element extracted from the data chunk, it must be deallocated in the destructor method of the iterator object.

– `input_coord` This output parameter holds the coordinates of the point, with which the input data element is associated, in the input attribute space.

The `genIterator` method of `T2_Dataset` may create a state when the iterator object is instantiated. The state information can be passed to the iterator object, and can be used to iterate through data elements at each call to the `getNextElement` method. There may be cases where no state information needs to be created. In those cases, the `getNextElement` method can use the `system` and `chunk` parameters (which are the same as corresponding parameters passed to `genIterator` method) to iterate through data elements.

The user-defined dataset object is instantiated in the constructor function implemented by the application developer. The signature of the constructor function is shown in Figure 10. The constructor function returns `T2_UDFRet_OK` on success or `T2_UDFRet_ERROR` on error.

- `system` This input parameter can be used to query system information such as number of processors and local processor id.
- `data_fid` This input parameter is an array of file descriptors for the data files of the dataset, stored on local disks.
- `meta_fid` This input parameter is an array of file descriptors for meta-data files on local disks.
- `dataset_obj` This output parameter is a pointer to the instance of the user-defined dataset object instantiated in the constructor function.

The iterator object may need the following information to function correctly: how many elements are there in a chunk?, what is the size of each element in bytes?, what is the data structure of an element and/or a data chunk? This information can be stored in different ways; along with each chunk, hard-coded into the dataset/iterator object, as a header information in data files, or in separate meta-data files. The `data_fid` and `meta_fid` parameters can be used to access data and meta-data files to extract the information needed by the dataset/iterator objects.

4.5 Data Aggregation Service

This service provides base classes to create and manipulate accumulator data structures, implement aggregation operations, and to convert accumulator values to the final output values.

4.5.1 Accumulator

An accumulator is a user-defined data structure to hold intermediate results during query processing. For example, an accumulator can be used to keep a running sum to compute an average as the output. Each accumulator is associated with a constructor function and two objects:

- *Accumulator meta-data object*. This object holds the accumulator meta-data. It also provides methods to strip mine the accumulator, i.e., partition the accumulator into tiles, so that each accumulator tile fits into the memory reserved for the accumulator on each processor.
- *Accumulator object*. This object encapsulates the data structures for an accumulator tile. It also provides methods to access individual accumulator elements.
- *Constructor function for accumulator meta-data object*. This function is used to instantiate an instance of user-define accumulator meta-data object.

An ADR query specifies an accumulator, and ADR invokes the constructor function, with arguments from the query specification, to create the accumulator meta-data object. The query planning service uses the methods of accumulator meta-data object to strip mine the accumulator so that each accumulator tile fits into the memory. Query execution service iterates through the query processing phases (see Section 2.5.2) one tile at a time, until all of the tiles have been processed. At the beginning of each iteration, ADR uses accumulator meta-data object methods to instantiate an accumulator object, which corresponds to the current tile. The ADR base classes for accumulator are shown in Figure 11.

The **T2_AccMetaObj** is the base class for accumulator meta-data objects. It provides the following virtual methods to be implemented by the application developer:

- `stripMine` This method is used to partition an accumulator into a set of tiles, so that each tile fits into the memory. It returns `T2_UDFRet_OK` on success or `T2_UDFRet_ERROR` if there is an error.
  - `query_info` This input object encapsulates information about the query and the machine ADR is running on, and can be used to get the query bounding box and to get system information such as the number of back-end nodes and local node id.
  - `mem_size` This input parameter holds the amount of the memory (in bytes) available for an accumulator tile. The size of each accumulator tile should be less than or equal to the value of `mem_size`.
  - `chunk_info_list` This input parameter is a list of meta-data (e.g., minimum bounding rectangle) for data chunks that intersect with the query. The chunk meta-data can be used to partition accumulator more efficiently. The coordinates of a minimum bounding rectangle (MBR) stored in `chunk_info_list` are defined in the *output attribute space*, not in the input attribute space. The MBR information for data chunks is generated using user-defined index (see Section 4.2) in the indexing service. The coordinates of each MBR is then projected to the output attribute space using the user-defined projection methods (`projectBox`) of the attribute space service (see Section 4.3).
  - `accum_rgn_list` This output parameter is a list of regions in the output attribute space. Each accumulator tile may be sparse data structure, and its pieces may be sparsely located in the output attribute space. Hence, each entry of `accum_rgn_list` is a list of bounding boxes of accumulator pieces in the corresponding accumulator tile.
- `allocAcc` This method is used to instantiate a user-defined accumulator object for the current accumulator tile. It returns `T2_UDFRet_OK` on success or `T2_UDFRet_ERROR` if there is an error.
  - `query_info` This input object encapsulates information about the query and the machine ADR is running on, and can be used to get the query bounding box and to get system information such as the number of back-end nodes and local node id.
  - `tile_id` This input parameter is the id of the accumulator tile to be created. If there are N accumulator tiles, the value of `tile_id` runs from 0 to $N-1$, and is consistent with the order of entries in `accum_rgn_list`. The accumulator meta-data object may store internal meta-data for each accumulator tile, the `tile_id` can be used to access the appropriate meta-data for the current accumulator tile.
  - `accum_rgn` This input parameter holds the MBR of the accumulator tile to be instantiated. The value of `accum_rgn` is calculated in `stripMine` and stored in the `accum_rgn_list` array.

```
#include "t2_acc.h"

// Accumulator meta-data object
class T2_AccMetaObj {
public:
  virtual T2_UDFRet stripMine(const T2_QueryInfo& query_info, size_t mem_size,
                              const T2_ClusterInfoList& chunk_info_list,
                              T2_VArray<T2_Region>& accum_rgn_list);
  virtual T2_UDFRet allocAcc(const T2_QueryInfo& query_info,
                             T2_Iteration tile_id, const T2_Region& accum_rgn,
                             T2_Accumulator*& acc_obj);
};

// Accumulator definition
class T2_Accumulator {
public:
  virtual T2_UDFRet navigateAll(const T2_QueryInfo& query_info,
                                T2_AccIterator*& accIter_obj);
  virtual T2_UDFRet navigate(const T2_QueryInfo& query_info,
                             T2_IteratorID accIter_id,
                             const T2_Region& output_rgn,
                             T2_AccIterator*& accIter_obj);
};

// Accumulator iterator
class T2_AccIterator {
public:
  virtual T2_UDFRet getNextElement(const T2_QueryInfo& query_info,
                                   T2_Accumulator& acc_obj,
                                   bool& end_of_data,
                                   void*& accum_data);
};

// Accumulator constructor function
typedef T2_UDFRet
(T2_AccMetaConstructor) (const T2_QueryInfo& query_info,
                         const T2_Box& output_box,
                         T2_UsrArgReader& user_arg,
                         T2_AccMetaObj*& accMeta_obj);
```

Figure 11: The ADR base classes to implement an accumulator.

  - `acc_obj` This output parameter points to the instantiated accumulator data structure.

The user-defined accumulator objects are derived from **T2_Accumulator** class. This class encapsulates the data structures of an accumulator tile, and provides methods to construct iterators to access accumulator elements.

- `navigateAll` This method constructs an accumulator iterator object that can be used to access all the elements of the accumulator tile. This iterator is used to initialize accumulator elements using the `aifElem` method of `T2_AggregateFuncObj`, described in the next section. It returns `T2_UDFRet_OK` on success or `T2_UDFRet_ERROR` if there is an error.
  - `query_info` This input object encapsulates information about the query and the machine ADR is running on, and can be used to get the query bounding box and to get system information such as the number of back-end nodes and local node id.
  - `accIter_obj` This output parameter points to the accumulator object instantiated in this method.
- `navigate` This method constructs an accumulator iterator object that can be used to access accumulator elements whose coordinates fall inside the region defined by `output_rgn`. It returns `T2_UDFRet_OK` on success or `T2_UDFRet_ERROR` if there is an error.
  - `query_info` This input object encapsulates information about the query and the machine ADR is running on, and can be used to get the query bounding box and to get system information such as the number of back-end nodes and local node id.
  - `accIter_id` This input parameter holds the id of the iterator. An accumulator object is allowed to have more than one iterator. The ADR query specifies which iterator is to be used for the given query.
  - `output_rgn` This input parameter can be used to construct an iterator that will access only the accumulator elements whose coordinates fall inside this region. The value of the `output_rgn` is computed in `project` method of the `T2_ProjectFuncObj` in attribute space service (see Section 4.3). It corresponds to the region in output attribute space to which an input point projects.
  - `accIter_obj` This output parameter points to the accumulator object instantiated in this method.

ADR uses the constructor function implemented by the application developer, to instantiate an instance of accumulator meta-data object. The signature of the constructor function is shown in Figure 11. The constructor function returns `T2_UDFRet_OK` on success or `T2_UDFRet_ERROR` on error.

- `query_info` This input object encapsulates information about the query and the machine ADR is running on, and can be used to get the query bounding box and to get system information such as the number of back-end nodes and local node id.
- `user_arg` This input parameter is the user-defined argument to the constructor function. It is passed from the application to the constructor function via the ADR query.
- `accMeta_obj` This output parameter points to the user-defined accumulator meta-data object instantiated in the constructor function.

4.5.2 Aggregation Operations

The aggregation functions are responsible for carrying out user-defined aggregation operations. ADR computes the output in four steps (see Section 2.5.2):

- *Initialization.* Once the accumulator object for the current accumulator tile is instantiated in each back-end node by the accumulator meta-data object, accumulator elements are initialized using methods in the aggregation object.
- *Local Reduction.* Data chunks are retrieved from the local disks. When a data chunk is in the memory and ready for processing, the input elements are projected to the corresponding accumulator elements, using `project` method in `T2_ProjectFuncObj`, and `navigate` method in `T2_Accumulator` classes. The input element is aggregated with the current values at the corresponding accumulator elements.
- *Global Combine.* After all of the input data chunks are processed, the accumulator elements in each processor is exchanged, and partial results stored in an accumulator element in one node is merged with partial results stored in the same accumulator element in other nodes. Note that at the end of this step, each back-end node may have a distinct subset of accumulator elements that contain the full intermediate values.
- *Output Handling.* The final output values are computed from the values in the accumulator elements.

ADR provides the **T2_AggregateFuncObj** base class for application developer to implement the processing carried out in these four steps. The declaration of the base class is shown in Figure 12.

- `aifElem` This method initializes an individual accumulator element. It returns `T2_UDFRet_OK` on success or `T2_UDFRet_ERROR` on error.
  - `query_info` This input object encapsulates information about the query and the machine ADR is running on, and can be used to get the query bounding box and to get system information such as the number of back-end nodes and local node id.
  - `accum_data` This parameter points to the accumulator element to be initialized.
- `aifAcc` This method initializes the entire accumulator object. It returns `T2_UDFRet_OK` on success or `T2_UDFRet_ERROR` on error.
- `dafElem` This method aggregates the input element to the corresponding accumulator element. It is called for all of the accumulator elements in the current accumulator tile, to which the input element projects. The iterator instantiated by `navigate` method of `T2_Accumulator` class is used to iterate through the accumulator elements. `dafElem` returns `T2_UDFRet_OK` on success or `T2_UDFRet_ERROR` on error.
  - `dataset_id` This input parameter is the id of the dataset from which the input element is accessed.
  - `elem_data` This input element points to the input data element. This pointer is set in `getNextElement` method of `T2_Iterator` object in attribute space service (see Section 4.3).
  - `accum_data` This parameter points to the accumulator element. The value of the accumulator element is updated in the method.

```
#include "t2_aggr.h"

class T2_AggregateFuncObj {
public:
  // virtual methods for accumulator initialization
  // initializing accumulator element
  virtual T2_UDFRet aifElem(const T2_QueryInfo& query_info, void *accum_data);
  // initializing entire accumulator
  virtual T2_UDFRet aifAcc(const T2_QueryInfo& query_info, T2_Accumulator& acc_obj);

  // virtual methods for local reduction phase
  virtual T2_UDFRet dafElem(const T2_QueryInfo& query_info, T2_DSID dataset_id,
                            const void* elem_data, void* accum_data);
  virtual T2_UDFRet dafAcc(const T2_QueryInfo& query_info,
                           const T2_Cluster& chunk,
                           const T2_Box& chunk_mbr,
                           T2_ClusterAuxInfoReader& chunk_meta,
                           const T2_Region& accum_rgn,
                           T2_Dataset& dataset_obj,
                           T2_DSID dataset_id,
                           T2_IteratorID dataset_iterID,
                           const T2_Box& query_box,
                           T2_ProjectFuncObj& proj_obj,
                           u_int acc_iterID, T2_Accumulator& acc_obj);

  // virtual methods for global combine phase
  virtual bool needGlobalCombine(const T2_QueryInfo& query_info,
                                 T2_Accumulator& acc_obj);
  virtual T2_UDFRet fillAccMsgBuffer(const T2_QueryInfo& query_info,
                                     T2_Accumulator& acc_obj,
                                     T2_Array<T2_AccMsgBufferWriter>& msgbuf);
  virtual T2_UDFRet processAccMsg(const T2_QueryInfo& query_info, T2_ProcID sender,
                                  T2_AccMsgBufferReader& msgbuf,
                                  T2_Accumulator& acc_obj);

  // virtual method for creating the output (output handling phase)
  virtual T2_UDFRet finalize(const T2_QueryInfo& query_info,
                             T2_Accumulator& acc_obj, T2_Output*& output_obj);
};

// Constructor function
typedef T2_UDFRet
(T2_AggregateFuncConstructor) (const T2_QueryInfo& query_info,
                     T2_UsrArgReader& user_arg, T2_AggregateFuncObj*& aggr_obj);
```

Figure 12: The ADR base class for aggregation operation.

- `dafAcc` This method takes the entire input data chunk and the accumulator object, and aggregates the values of input elements in the data chunk with the values in the accumulator elements. ADR provides a default implementation for this method using methods implemented in various services– note that the corresponding objects are passed as parameters to `dafAcc` method. However, user can override the ADR implementation with a more efficient, application specific implementation. `dafAcc` returns `T2_UDFRet_OK` on success or `T2_UDFRet_ERROR` on error.

- `needGlobalCombine` This method returns `true` if the global combine step is required to compute final accumulator values. It returns `false` otherwise.

- `fillAccMsgBuffer` This method stores the accumulator elements in a node into message buffers destined for other nodes. It returns `T2_UDFRet_OK` on success or `T2_UDFRet_ERROR` on error.

  - `msgbuf` This output parameter encapsulates a list of message buffers. Each entry corresponds to a message buffer destined for a back-end node (the local node itself is also included). The set of accumulator elements to be sent from the local node to another back-end node is written into the corresponding message buffer.

- `processAccMsg` This method processes a message buffer received from another back-end node. The data values in the message buffer are unpacked and aggregated with the corresponding accumulator elements stored in the local node. `processAccMsg` returns `T2_UDFRet_OK` on success or `T2_UDFRet_ERROR` on error.

  - `sender` This input parameter holds the id of the back-end node that has sent the message buffer.
  - `msgbuf` This input parameter encapsulates the data values received from the `sender` node.
  - `acc_obj` This parameter is a reference to the accumulator object in the local node.

- `finalize` This method constructs an output object from the accumulator object after global combine phase is completed.

  - `acc_obj` This input parameter is a reference to the accumulator object. The values of the accumulator elements in this accumulator object is used to create the output data values.
  - `output_obj` This output parameter points to the output object instantiated in this method.

4.5.3 Final Output

The output is a data structure that is expected as a result of the query. It is generated from the accumulator. A single output object on a back-end node usually represents one part of a distributed data structure partitioned across all of the back-end nodes. Therefore, the necessary information should be stored with the output object so that application client can reassemble the output. ADR provides the **T2_Output** base class that encapsulates the output data structure and has methods to write output into an ordered list of contiguous buffers. ADR sends these buffers to the receiving client in the order specified in the list. The definition of the ADR base class is shown in Figure 13.

```
#include "t2_output.h"

class T2_Output {
public:
  virtual size_t getOutputBufferSize(const T2_QueryInfo& query_info);
  virtual T2_UDFRet flushOutput(const T2_QueryInfo& query_info, T2_OutputBuffer& buf,
                                T2_OutputDataPtrList& ptr_list);
};
```

Figure 13: The ADR base class for creating the final output to be sent to the client.

- `getOutputBufferSize` This method returns the amount in bytes of additional buffer space needed to pack a part of the output.
- `flushOutput` This method creates an ordered list of `(void*, size)` tuples, each of which points to a contiguous buffer and stores its size. The contiguous buffers are sent to the client.
  - `query_info` This input object encapsulates information about the query and the machine ADR is running on, and can be used to get the query bounding box and to get system information such as the number of back-end nodes and local node id.
  - `buf` This parameter is a reference to a contiguous buffer space allocated by ADR to pack some output data. The size of this buffer is returned from `getOutputBufferSize`.
  - `ptr_list` This parameter is an ordered list of pointers to contiguous buffers and their sizes. The buffers, which are pointed to in this list, are sent to the client.

`T2_OutputDataPtrList` is a list of pairs of data pointers and data sizes, and allows new pairs to be appended to the end of the list. ADR is responsible for sending the buffers in the specified order back to the requesting client. The additional buffer space specified by `getOutputBufferSize` is used by method `flushOutput` to put the output into contiguous memory locations. For example, an output implemented as an array often carries some additional information for the entire array, such as the starting position of the starting pixel in an image array. It may also be convenient if some kind of encoding needs to be performed before sending the output data back to the client. If all information that needs to be sent to the client is already in contiguous space, the output can decline any additional memory space and simply return a list of pointers pointing to the real data.

Having a data type for the output data separate from the accumulator allows a cleaner interface. Without an explicit definition of the final output, all the functions needed for the output will be pushed into the aggregation function. Having an explicit function, however, allows a clean interface for the aggregation function, as well as a clean interface between the query and the client that is expecting the final output. The querying client just needs to know that it is able to parse whatever the output data type sends across the wire. For aggregation functions, such as an average, that actually generates a different data structure from that of the accumulator, this is necessary. For aggregation functions that can reuse the space of the accumulator to store the final output, such as taking the *min* or *max*, no additional copy is needed as long as the accumulator supports the interface that the output data type (i.e. `T2_Output`) requires.

5 Loading Datasets into ADR

5.1 Overview

The data loading service manages the process of loading new datasets into ADR. To achieve low latency retrieval of data, a dataset is partitioned into a set of chunks, each of which consists of one or more data items. A chunk is the unit of I/O and communication in ADR. That is, a chunk is retrieved as a whole during query processing. As every data item is associated with a point in a multi-dimensional attribute space, every chunk is associated with a minimum bounding rectangle (MBR) that encompasses the coordinates (in the associated attribute space) of all the items in the chunk. Since data is accessed through range queries, it is desirable to have data items that are close to each other in the multi-dimensional space in the same data chunk. Data chunks are declustered across the disks attached to back-end nodes to achieve I/O parallelism during query processing. Data chunks on a single disk are clustered to obtain high bandwidth from each disk. An *ADR index*, constructed from the MBRs of the chunks, is used to find the chunks that intersect a query window during query processing.

Loading a dataset into ADR is accomplished in four steps:

1. partition the dataset into data chunks,
2. compute placement information for the data chunks,
3. create an ADR index,
4. move data chunks to the disks according to placement information,
5. update the dataset and index catalogs of the ADR back-end.

The placement information describes how data chunks are declustered and clustered across the disk farm. The result of the loading process is an ADR dataset, which consists of a set of ADR dataset files and a set of index files. Each ADR dataset file contains a set of data chunks, each of which has a corresponding entry in one of the index files. The entry records the MBR and the size of the data chunk, the dataset file that contains the data chunk, and the offset into the dataset file for the data chunk. After data chunks are stored in the disk farm, a *dataset registration process* is used to update the dataset and index catalogs, after which the new dataset becomes available to the ADR back-end. Section 6.6 describes in details the dataset registration process.

A dataset to be loaded into ADR is referred to as a *source dataset*. It is specified as a set of source data files that contain the data items, along with meta-data information. For the purpose of dataset loading, we categorize source datasets into three classes.

1. A *fully cooked* source dataset is already partitioned into data chunks, and the chunks have been declustered across a set of dataset files. That is, the source data files associated with the dataset to be loaded contain data chunks, and therefore can be used directly as ADR dataset files. A corresponding index (e.g., an R-tree [12]) has also been created. Only step 4 and 5 must be executed for fully cooked datasets. A user can load a fully cooked source dataset by first moving the source dataset files and index files to the disks accessed by ADR back-end, and then registering the dataset using the ADR dataset registration program described in Section 6.6.

2. A *half cooked* source dataset is already partitioned into data chunks as for a fully cooked dataset. However, no effort has been made to layout the data chunks across the source data

files for better I/O performance. One or more meta-data files are used to store for every data chunk its MBR, the half cooked file that contains the data chunk, and a start and end position for the data chunk in the half cooked file. The ADR data loading service uses MBRs from the meta-data files to compute new placement information for the data chunks so that better I/O performance can be achieved at query processing time. It then moves the data chunks according to the newly computed placement information, and builds an R-tree index. The ADR data loading service also generates a summary file, which together with other information is used during the dataset registration process.

3. For a *raw* source dataset, no placement information is pre-computed, and the dataset has not been partitioned into data chunks. Loading a raw source dataset requires going through all five steps listed above.

The data loading service currently provides support for half cooked source datasets. We plan to design an interface for raw source datasets in the future. This new interface will allow users of raw source datasets to]incorporate user-defined partitioning methods into the data loading service.

5.2 The ADR Data Loader

The ADR data loader utility program loads a half-cooked source dataset into ADR. It consists of a *master manager* and *data movers*. The master manager reads the meta-data for the source dataset, and computes placement information for the data chunks. ADR uses a declustering method based on *Hilbert curve* [9] to enable fast declustering and clustering of large datasets. A fast placement method is especially useful when datasets are permanently stored in archival (i.e. tertiary) storage devices, so that ADR must use its local disks as a cache to load datasets on demand.

Data movers are responsible for actually copying the data chunks onto disks in the ADR back-end. The data mover running on each of the back-end nodes is responsible for storing the data chunks to the disks attached to that node. After the master manager computes the placement information, it broadcasts the information to the data movers. Each data mover extracts placement information for data chunks that must be placed on its local disks. Each mover accesses the source data files for the source dataset and copies the required data chunks onto its local disks. After all data chunks are copied, each data mover builds an *ADR index* on its local data chunks. The ADR index is used by the back-end process running on that node during query execution. The ADR indexing service uses an R-tree implementation based on the Gist C++ library [21] developed at the University of California, Berkeley.

The current implementation of the data loading service requires that each mover be able to access all the source data files of the dataset to be loaded. In addition, the data loader does not perform subsetting of a large dataset so that only a portion of the dataset is loaded into ADR.

To load a half cooked dataset into ADR, the user has to provide a *back-end configuration* file and a *loader command* file.

A **back-end configuration file** is an ASCII file that describes the set of back-end nodes, the disk farm of the parallel machine that the ADR back-end runs on, and the connectivity information between the ADR back-end nodes and disks. ADR considers the disk farm of the parallel machine it runs on as a set of *logical disks*. The back-end configuration file assigns each logical disk to a back-end node, and during query processing time, a back-end node is responsible for reading data chunks from all dataset files assigned to its assigned logical disks. The back-end configuration also assigns each logical disk to a physical disk in the disk farm, while a physical disk can be assigned multiple logical disks. As to be seen later, dataset files for loaded ADR datasets are assigned to

logical disks. The abstraction through logical disks is helpful when dataset files stored on the same physical disk must be assigned at run-time to two or more back-end nodes to achieve better load balance. A back-end configuration file consists of a list of entries, one per back-end node. The format for such an entry is given in Figure 14. See Section 6.6 for more details. Note that in the back-end configuration, each back-end node is given a unique logical processor id, each physical disk is given a unique physical disk id, and each logical disk is given a unique logical disk id.

A **loader command file** is an ASCII file, which contains a set of data loader commands, one for loading a half cooked source dataset. A data loader command specifies the following information.

1. A data loader command specifies a dataset name, which will be used in the output summary file as the dataset name for the loaded ADR dataset. This name can also be used as the dataset name during the dataset registration process.

2. A data loader command also specifies the set of logical disks over which the data chunks of the half cooked source dataset are declustered over, along with the prefixes to be used for the dataset files and the index files over those logical disks. The ADR data loader generates one dataset file for each of the specified logical disk, and the file would be named using the given dataset file prefix and the logical disk id. The ADR data loader also generates an index file for each dataset file on the same logical disk that the dataset file is assigned to, and the index file would be named using the given index file prefix and the logical disk id.

3. A data loader command specifies a list of *name file/linear index file* pairs. A name file contains a list of source data files for the source dataset being loaded. These files contain the data chunks to be loaded into ADR. A linear index file contains for each data chunks to be loaded its MBR and its location in one of the source data files listed in the corresponding name file. It can also contain application-dependent data, referred to as *user data*, for each data chunk. User data of a data chunk is inserted into the ADR index as user defined meta-data for a data chunk, and is retrieved during query processing for all data chunks that intersect the given query. The linear index file can be an ASCII or a binary file. In an ASCII format, user data for a data chunk cannot exceed a single line. In a binary format, user data is preceded by the number of bytes for the user data.

   Note that there is one-to-one correspondence between name files and linear index files. That is, one linear index file contains information for data chunks stored in source data files listed only in one name file. Figure 15 and Figureřeflinear-index-fig show the formats for a name file and a linear index file, respectively.

Figure 17 shows the format for a data loader command.

Figure 18 illustrates the process of loading half cooked source datasets into ADR. Loading of a set of source datasets is accomplished in the following steps: First, all movers read back-end configuration file to extract information about the back-end nodes and the disks. Second, the master manager reads a command from the loader command file to retrieve information about the source dataset to be loaded. Third, the master manager reads the linear index files listed in the command file and computes placement information for the data chunks of the source dataset. Fourth, it broadcasts placement information and source dataset information to the mover processes. In the current implementation, one of the mover processes also functions as a master manager, i.e., there is no separate master manager process. Finally, after receiving placement and source dataset information, each mover process accesses source data files, copies data chunks onto the local disks according to the computed placement information, and builds an ADR index for data chunks on local disks.

```
# The back-end configuration file.
#
# There is one entry per back-end node, and each entry has the following
# format.
#     char[] hostname:   hostname for the back-end node
#     u_int logical_proc_id: the logical processor id for the back-end node
#     <information for disk 1>
#     <information for disk 2>
#           :
#     ===           : end of entry mark
#
# <information for disk x> can be a record in the following format
# for a local disk connected directly to the back-end node,
#      'l'      : the label for a local disk record
#      u_int physical_disk id
#      <a list of disk id ranges>: the set of logical disks assigned to this
#                        physical disk
#      -1
# or in the following format for a  set of remote disk that is
# cross-mounted on this back-end node,
#      'r'      : the label for a remote disk record
#      <a list of disk id ranges>: the set of physical disks cross-mounted
#                        on this back-end node
#      -1
# where <a list of disk id ranges> consists of a list of ranges,
# each of which is in one of the two following formats.
#     u_int n: a single disk id
#     n-m: all disk id's from n up to m

host1  0          # hostname and logical processor id
l 0    0 -1       # physical disk 0 is a local disk, assigned logical disk 0
r      1-3  -1    # local host can access remote physical disk 1,2,3
===
```

Figure 14: The back-end configuration file format. The text after a # is considered a comment.

```
# The name file.
#
# char filename1[]
# char filename2[]
# ...
#
path/data-file.1  # name of the data file
path/data-file.2
.
.
.
path/data-file.N  # name of the data file
```

Figure 15: The name file format. Text after a # is considered a comment.

```
# The linear index file.
#
# <header information>
# <information for entry 1>
# ...
# <information for entry N>
#
# < header information>:
#   char format: format of the linear index file, 'a' for ASCII, 'b' for binary
#   unsigned int ndims: number of dimensions of the underlying
#                       multi-dimensional attribute space
#   unsigned int nentries: number of entries in the file
# < information for entry x>:
#     float low[ndims]: coordinates of minimum point of the minimum bounding
#                       box of the data chunk
#     float high[ndims]: coordinates of maximum point of the minimum bounding
#                        box of the data chunk
#     unsigned int nblocks: number of physical file blocks that constitute a
#                           data chunk.
#     <information for block 1 of entry 1>: information about physical block
#     <information for block 2 of entry 2>: information about physical block
#     ...
#     <user data> : user-defined data
# < information for block y of entry x>:
#     unsigned int file_id: line number of data file listed in the name file.
#     off_t offset: offset into the data file
#     size_t size: size of the physical block in bytes
# < user data>:
#     size_t nbytes: length of the user-defined data
#     char usrdata[nbytes]: user-defined data. Length of usrdata array cannot
#                           exceed a single line for ASCII index file.
#
format ndims nentries             # <header information>
low[0]  low[1]  ... low[ndims-1]  # <information for entry 1>
high[0] high[1] ... high[ndims-1]
nblocks
file_id offset size               # <info for physical block 1 of entry 1>
file_id offset size               # <info for physical block 2 of entry 1>
...
nbytes                            # <user data for entry 1>
usrdata[nbytes]
low[0]  low[1]  ... low[ndims-1]  # <information for entry 2>
high[0] high[1] ... high[ndims-1]
nblocks
file_id offset size               # <info for physical block 1 of entry 2>
file_id offset size               # <info for physical block 2 of entry 2>
...
nbytes                            # <user data for entry 2>
usrdata[nbytes]
...
```

Figure 16: The linear index file format. In an ASCII linear index file, text after a # is considered a comment.

```
# The loader command file contains a set of data loader commands,
# one per source dataset to be loaded, and each data loader command
# has the following format.
#     char[] dataset_name:   name of the dataset after being loaded
#     <disk and file prefixes 1>
#     <disk and file prefixes 2>
#            :
#     -1
#     <name file and index file 1>
#     <name file and index file 2>
#            :
#     ===        - end of command mark
#
# where <disk and file prefixes x> has the following format.
#     <logical disk id range>         -- one range of logical disk id's
#     char[] dataset_file_prefix           -- prefix for the dataset files
#                                       assigned to the logical disk id's
#                                       listed in the logical disk id range
#     char[] index_file_prefix             -- prefix for the index files
#                                       assigned to the logical disk id's
#                                       listed in the logical disk id range
#
# and <logical disk id range> is in one of the following formats:
#    n                       - a single logical disk id
#    n-m                     - a set of logical disk id's from n upto m
#
# and <name file and index file> is in the following format:
#    char[] name_file        - filename of a name file
#    char[] index_file       - filename of an index file
#

dataset-name-1
0        disk-0/data-file        disk-0/index-file    # data file and index file
1        disk-1/data-file        disk-1/index-file    # data file and index file
2        disk-2/data-file        disk-2/index-file    # data file and index file
3        disk-3/data-file        disk-3/index-file    # data file and index file
-1
sample.name               # name file for data subset 1 on tapes
sample.idx               # index file for data subset 1 on tapes
                        # more name files and index files can be added here
===
...
```

Figure 17: The loader command file format. The text after a # is considered a comment.

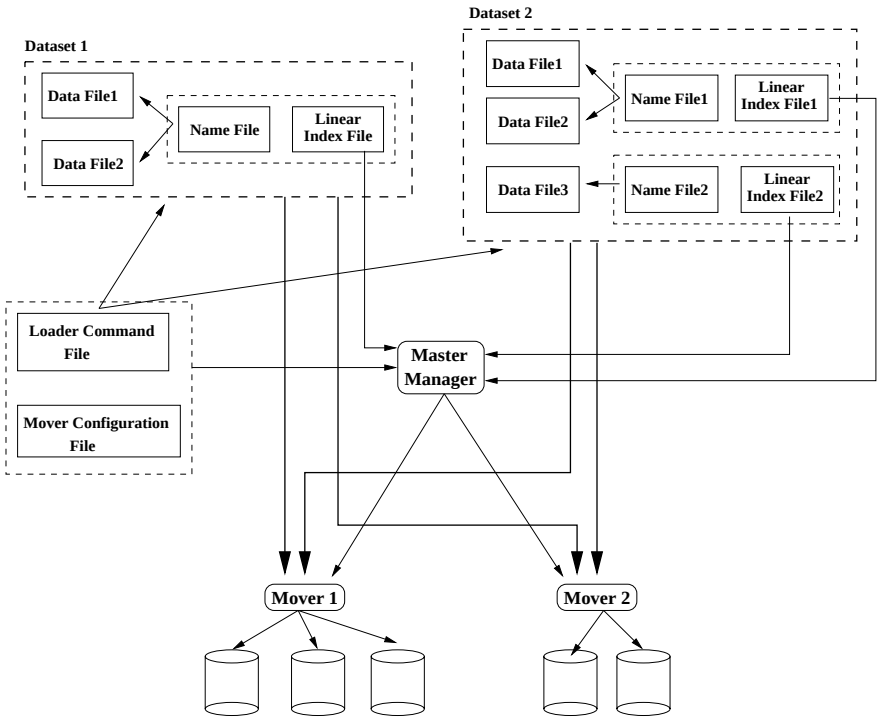

Figure 18: Loading half cooked datasets into ADR.

6 Installing and Running ADR

The ADR source distribution contains the following files and directories.

- `COPYRIGHT.TXT`, which describes the copyright information regarding the ADR source distribution,
- `README`, which describes the files and directories in this distribution, and steps to install ADR,
- `Version`, which contains the date for this release,
- `arch/`, which contains the machine configuration files created by the ADR system configuration utility program,
- `back-end/`, which contains the source code for the parallel ADR back-end,
- `bin/`, which contains the executables for ADR utility programs and the ADR front-end,
- `common/`, which contains the source code for classes shared by the ADR back-end and the ADR front-end,
- `doc/`, which contains documentation of ADR, including this document,
- `example/`, which contains the customization code for example applications implemented with ADR,
- `front-end/`, which contains the source code for the ADR front-end,
- `lib/`, which contains the ADR libraries and object files generated after compilation,
- `meta-chaos/`, which contains the source code for implementing the interface that uses Meta-Chaos [8] to transfer the query results to a parallel program (documentation on this interface will be included in the future),
- `rtree/`, which contains the source code for an R-tree implementation based on the Gist C++ library [21] developed at the University of California, Berkeley,
- `sback-end/`, which contains the source code for a sequential version of the ADR back-end, which can be used for debugging customization code,
- `utility/`, which contains the source code for ADR utility programs.

Installing ADR consists of the following steps.

1. Configure the ADR library (Section 6.1).
2. Compile the ADR library and various ADR utility programs (Section 6.2).

At the end of a successful installation process, machine configuration files are generated in the `arch/` directory, ADR libraries are generated in the `lib/` directory, and executables for the ADR front-end and various utility programs are generated in the `bin/` directory. Once the installation process completes, customization code for the ADR back-end can be implemented, as described in Section 4. Compiling a complete parallel ADR back-end executable with the customization code is done through the following steps.

1. Compile customization code (Section 6.3).

2. Register user-defined constructor functions with ADR (Section 6.4).

3. Link ADR libraries with the customization code (Section 6.5).

As a result of a successful compilation process, an executable for the customized parallel ADR back-end is generated. After datasets are loaded (see Section 5) and registered (see Section 6.6) with ADR, the full application with the ADR front-end and the customized parallel ADR back-end can be started, as described in Section 6.7.

The ADR installation process relies on macro substitution in the following form,

$$\text{MACRO1} = \$(\text{MACRO2:op\%os=np\%ns})$$

If the utility program "make" on the target machine does not support such substitution, then "gmake" from GNU can be used instead. In the remaining of this section, we will assume that the utility program "make" on the target machine does support the desired macro substitution.

6.1 Configuring the ADR Library

The purpose of the configuration process is to correctly set certain architecture-dependent compilation options, and to locate certain supporting libraries (such as MPI) so that the ADR library can be correctly compiled on the target machine. Many of these options have default values, and the ADR configuration program uses default values for many of these options. However, if desired, a user can provide the appropriate configuration option values to override the ADR default values by editing the user configuration file, `config.in` in the ADR root directory. This file contains the set of valid options for configuring the ADR installation. These options affect how the ADR codes are compiled. Many options have default values, so a user only needs to include in this file the options whose default values are to be overridden. Note that the ADR distribution consists of both sequential executables and parallel executables. Some of the options listed in this file affect the compilation of both the sequential and the parallel codes, whereas other options only affect either the sequential or the parallel codes. Each line in file `config.in` specifies the value for a configuration option, and it has the following format.

```
<option name>[= <option value> [<option value> ...]]
```

For those options that require values, their values must be given in one line. Texts after "#" in the file are considered as comments and are ignored. Figure 19, 20, 21 list the valid ADR configuration options and their default values. Some of these options are briefly explained below.

- `comm` defines the inter-processor communication interface to be used by the back-end nodes; the current implementation only uses the MPI interface, therefore `mpi` is the only valid value for this option,

- `mpidir`, `mpiinclude` and `mpilib` specify paths to the MPI include directory and the MPI libraries,

- `aio` defines the asynchronous I/O interface that the ADR back-end uses to retrieve data chunks from disks,

option name	option value	default value
comm	mpi	mpi
mpidir	<path to MPI root directory>	aix: others: $MPI_ROOT
mpiinclude	<MPI include directory>	aix: others: <mpidir>/include
mpilib	<MPI libraries>	aix: others: -L<mpidir>/lib/<mpiarch>/ch_p4 -lmpi
aio	<asynchronous I/O interface>	aix: aix sunos: sunos others: posix
mc		(Meta-Chaos interface disabled)
pvmdir	<path to PVM root directory>	$PVM_ROOT
pvmarch	<PVM architecture>	`<pvmdir>/lib/pvmgetarch`
pvminclude	<PVM include directory>	<pvmdir>/include
pvmlib	<PVM libraries>	-L<pvmdir>/lib/<pvmarch> -lgpvm3 -lpvm3 -lfpvm3
dclockname	<a timing function>	(default ADR timing function)
dclockpath	<.o file for <dclockname>>	

Figure 19: Configuration options for specifying communication interface, asynchronous I/O interface, and timing function for ADR compilation. MPI_ROOT and PVM_ROOT are shell environment variables.

- `mc` enables the interface that sends query output to (parallel) programs through the Meta-Chaos interface; this interface is disabled by default,
- `pvmdir`, `pvmarch`, `pvminclude` and `pvmlib` specify paths to the PVM include directory and PVM libraries, which are used only when the Meta-Chaos output-handling interface is enabled,
- `dclockname` and `dclockpath` allow a user to override the default function that ADR uses to measure wall-clock time; `dclockname` provides the function name for the user-defined timing function (a mangled function name if implemented in C++), and `dclockpath` specifies the .o file that contains the implementation for the user-defined timing function,
- `frontenddef`, `pbackenddef` and `sbackenddef` affect the behaviors of the ADR front-end, the ADR parallel back-end, and the ADR sequential back-end through the following values:

  `-DT2_ASSERTION` enables various assertion tests during execution,

  `-DT2_VERBOSE[=n]` enables output messages being printed to the screen during execution,

  `-DT2_LABEL_IO` makes each ADR parallel back-end node to precede its output message printed to the screen with its MPI processor rank,

  `-DT2_GETRUSAGE` makes the ADR parallel back-end to report information obtained from the system call `getrusage` after query execution,

  `-DT2_TIME_PLANNING[=n]` enables the ADR parallel back-end to report for each received batch of queries the amount of time spent in the query planning service ,

  `-DT2_TIME_EXECUTE[=n]` enables the ADR parallel back-end to report for each received batch of queries the amount of time spent as a group in the query execution service ,

option name	option value	default value
flag hline include	<compilation flags> <include directories>	
cc	<sequential C compiler>	aix: xlc others: gcc
ccflag	<C compilation flags>	
ccinclude	<C include directories>	
cpp	<sequential C++ compiler>	aix: xlC others: g++
cppflag	<C++ compilation flags>	
cppinclude	<C++ include directories>	
pcc	<parallel C compiler>	aix: mpcc others: gcc
pccflag	<C compilation flags>	
ccinclude	<C include directories>	
pcpp	<parallel C++ compiler>	aix: mpCC others: g++
pcppflag	<C++ compilation flags>	
pcppinclude	<C++ include directories>	
ld	<linker for sequential code>	<cpp>
ldflag	<C++ compilation flags>	
lib	<libraries to link with sequential code>	aix: sunos: -laio -lsocket -lnsl -lresolv -lbind linux: -lrt others:
pld	<linker for parallel code>	<pcpp>
pldflag	<C++ compilation flags>	
plib	<libraries to link with parallel code>	aix: sunos: -laio -lsocket -lnsl -lresolv -lbind linux: -lrt others:
ar	<archiver>	ar
arflag	<archiver flags>	ruv
ranlib	<archive randomizer>	aix: ranlib others:

Figure 20: Configuration options for compiling the ADR code.

option name	option value
frontenddef	-DT2_ASSERTION -DT2_VERBOSE[=n]
pbackenddef	-DT2_ASSERTION -DT2_VERBOSE[=n] -T2_LABEL_IO -DT2_GETRUSAGE -DT2_TIME_PLANNING[=n] -DT2_TIME_EXECUTE[=n] -DT2_TIME_QUERY[=n]
sbackenddef	-DT2_ASSERTION -DT2_VERBOSE[=n]

Figure 21: Configuration options for affecting the behaviors of the ADR front-end, the ADR parallel back-end, and the ADR sequential back-end.

`-DT2_TIME_QUERY[=n]` enables the ADR parallel back-end to report for each received query the amount of time spent in the query execution service .

For an option that can be assigned a value `n`, we have $1 \leq n \leq 5$. The larger `n` is, the more detailed information is printed.

Figure 22 shows an example of the file `config.in`. The default values for all options can be restored using the following command.

```
% make default
```

```
# config.in

mpiinclude=     -I/usr/local/mpi/include
mpilib=         -L/usr/local/mpi/lib -lmpi

flag=           -O2

pbackenddef=    -DT2_ASSERTION -DT2_LABEL_IO -DT2_VERBOSE=1
```

Figure 22: An ADR configuration file example.

To start the configuration process, use the following command at the system prompt.

```
% make config
```

The result of the configuration process will be stored in several files in the `arch/` subdirectory.

6.2 Compiling the ADR Library and Utility Programs

To compile the ADR library and various utility programs, simply use "make install" in the ADR root directory after performing the ADR configuration process as described in Section 6.1. This will create the following files in the `lib/` directory.

- `pbe.o` : the main program for the parallel ADR back-end program.
- `libt2be.a` : the library to link with when compiling the parallel ADR back-end program.
- `libt2fe.a` : the library to link with when compiling the application front-end.
- `sbe.o` : the main program for the sequential ADR back-end program.
- `libt2sbe.a` : the library to link with when compiling the sequential ADR back-end program.

The compilation process also creates the following executables in the `bin/` directory.

- `t2fe`, the ADR front-end program,
- `register-constructors`, the utility program to register user-defined constructor functions with ADR (see Section 6.4),
- `register-datasets`, the utility program to register datasets with ADR (see Section 6.6),

Section 6.7 describes how to run the ADR front-end program.

6.3 Compiling the Customization Code

To compile the customization code, one needs to include the appropriate ADR header files. To compile the customization code for the parallel ADR back-end, add the directory `common/include/` and `back-end/include/` to the list of directories searched for header files during compilation. To compile the customization code for the application front-end, add the directory `common/include/` and `front-end/include/` to the list of directories searched for header files during compilation. Figure 23 summarizes the compilation options to use when compiling the customization code, assuming that the variable ADR_DIR contains the path to the ADR root directory. To compile customization code that uses the ADR R-tree implementation, Add the directory `rtree/` to the list of directories for header files.

customization code for	compilation options
parallel ADR back-end	-I$(ADR_DIR)/common/include -I$(ADR_DIR)/back-end/include
application front-end	-I$(ADR_DIR)/common/include -I$(ADR_DIR)/front-end/include
uses R-tree	-I$(ADR_DIR)/rtree

Figure 23: Compilation options when compiling the customization code. The options here assume that the variable ADR_DIR stores the path to the ADR root directory.

6.4 Registering Constructor Functions

The ADR internal services interact with the user-defined customization Classes through the interface defined by the various ADR base classes. However, creating an instance of a customization class, in many cases, is done through the invocation of a user-defined "constructor function". In particular, the customization classes for dataset objects, index objects, projection function objects, accumulator meta-data objects, and aggregation function objects must be accompanied by their constructor functions. These functions will be called by the ADR internal services at run-time to create instances of the appropriate customization classes as needs arise.

Constructor functions must be registered with ADR before they can be used by the ADR services. The purposes of registration are two-fold:

1. ADR creates a code segment to keep function pointers to all the constructor functions. At run-time, ADR services would invoke these constructor functions through these function pointers.

2. ADR assigns ids to constructor functions, and the id's are used to "name" the appropriate constructor functions when formulating ADR queries.

Registration of constructor functions is done through an ADR utility program, `register-constructors`, which is generated during the ADR installation process. This utility program reads from a *constructor registration file*, and generates two output files: a C++ code segment that sets function pointers to the registered constructor functions, and a *constructor catalog file*. `register-constructors` takes three command-line arguments.

```
% register-constructors <constructor registration file>  \
                        <C++ code segment file> <constructor catalog file>
```

The generated C++ code segment must be compiled and linked with when generating the parallel ADR back-end executable, `pbe`. The generated constructor catalog is used by the ADR front-end to answer inquiries about the constructor functions. Figure 24 shows an example of the constructor catalog with five entries. Each entry in the constructor catalog file contains a function type, a function id, a function name, and a one-line description of the function. An entries for a constructor function of a dataset object also contain the number of dataset iterators provided by the datset object, and the names of those dataset iterators. Functions of the same type cannot share the same name, and cannot be assigned the same function id. The utility program `register-constructors` would complain if violation is detected.

A constructor registration file, which is created by the user, contains the necessary information for all constructor functions to create the C++ code segment file and the constructor catalog. Figure 25 shows an example of the constructor registration file. An entry in the constructor registration function contains the following fields, listed in the order they appear in the file.

1. a function type: this can be one of the following:

   - *accumulator* (or *ac* in short) for accumulator meta-data object constructor function
   - *aggregation* (or *ag* in short) for aggregation function object constructor function
   - *projection* (or *pr* in short) for projection function object constructor function
   - *dataset* (or *da* in short) for dataset object constructor function

```
# constructor catalog

accumulator             0
t2-svm-example:image-accumulator-meta-constructor
This is the accumulator meta constructor for T2's svm example.

aggregation             0
t2-svm-example:aggregation-function-max-constructor
The aggregation function that keeps the max-val pixel for T2's svm example.

projection              0
t2-svm-example:projection-function-constructor
The projection function for T2's svm example.

dataset         0
t2-svm-example:image-dataset-constructor
The image dataset for T2's svm example.
1 iterator-1  # iterators

index           0
t2-svm-example:image-dataset-index-constructor
The index for an image dataset for T2's svm example.
```

Figure 24: An ADR constructor catalog file example.

- *index* (or *in* in short) for index object constructor function

2. a function name: a unique name for the function
3. the actual C++ function name: this is the name of the C++ function that implements the constructor function
4. a one-line description of the function: where an empty line right the C++ function name means no description
5. for a function of a dataset object, the entry also contains the number of dataset iterators provided by the datset object, and the names of those dataset iterators.

Text in a line preceded by "#" is considered comment and will be ignored by `register-constructors`.

Each constructor function is expected to be given a unique name. The utility program `register-constructors` fails if it sees names that are not unique. Names are used by application front-ends to inquire of the ADR front-end for their function id's, which are used when formulating ADR queries. In the case where an ADR instance is customized for multiple applications and each application has its own set of constructor functions that are not applicable for another application, users are encouraged to assign unique names to the applications, and use the application names as prefixes for the names of their constructor functions in a format such as the following.

```
<application name>:<function name>
```

This not only avoids name clashing between functions from different applications, it also allows an application front-end to quickly inquire for the constructor functions implemented for its application using the ADR front-end inquiry interface that uses pattern matching over regular expressions. For example, the function names in Figure 25 are functions for the SVM application that is available with the ADR code distribution. The SVM front-end can find out all the SVM constructor functions simply by inquiring for functions with names that start with "`t2-svm-example:`".

Note that since multiple datasets can share the same dataset constructor function, datasets are not accessed by their dataset constructor function id's. Therefore, an application front-end almost never needs to know the function id for a dataset constructor function. This is also the case for indexes. Therefore, the ADR front-end inquiry interface does not return information about dataset constructor functions and index constructor functions. Access to datasets and their associated indexes are done through dataset id's and index id's, which are assigned by the dataset registration process to be discussed in the next section.

```
# constructor registration file

accumulator
t2-svm-example:image-accumulator-meta-constructor
svmImageAccMetaConstructor
This is the accumulator meta constructor for T2's svm example.

aggregation
t2-svm-example:aggregation-function-max-constructor
svm_aggregationMaxConstructor
The aggregation function that keeps the max-val pixel for T2's svm example.

projection
t2-svm-example:projection-function-constructor
svm_ImgPrjConstructor
The projection function for T2's svm example.

dataset
t2-svm-example:image-dataset-constructor
svm_ImageDatasetConstructor
The image dataset for T2's svm example.
1 iterator-1

index
t2-svm-example:image-dataset-index-constructor
svm_ImageIndexConstructor
The index for an image dataset for T2's svm example.
```

Figure 25: An ADR constructor function registration file example.

6.5 Linking with ADR Libraries

The executable for the ADR front-end, `bin/t2fe`, is generated when the ADR libraries are compiled (see Section 6.2). To generate an executable for the application front-end, the application code must

link with the library `lib/libt2fe.a`, which provides implementation of the ADR services described in Section 3.

Generating the executable for the parallel ADR back-end loaded with the customization code requires the linkage of `lib/pbe.o`, which provides the main program for the parallel back-end, all the object files obtained by compiling the customization code, and the library `lib/t2be.a`. Generating the executable for the sequential ADR back-end loaded with the customization code requires the linkage of `lib/sbe.o`, which provides the main program for the sequential back-end, all the object files obtained by compiling the customization code, and the library `lib/t2sbe.a`. Section 6.7 describes how to run the ADR back-end.

6.6 Registering Datasets

Just like constructor functions need to be registered with ADR, datasets also need to be registered with ADR. Registration of a dataset informs ADR of the dataset files of the dataset, and the index files of the ADR indexes associated with the dataset. It also describes how dataset files are assigned to the back-end nodes. The parallel ADR back-end considers the parallel machine that it runs on consists of a number of *logical disks*. A logical disk is assigned to a physical disk of the parallel machine, and a physical disk can be assigned multiple logical disks. This abstraction is helpful when dataset files stored on the same physical disk are assigned at run-time to two or more back-end nodes to achieve, for example, better load balance. A *back-end configuration file* is used by ADR to describe the relationship between back-end nodes and disks. Figure 26 shows an example of the back-end configuration file. Section 5.2 has described how the back-end configuration file is used by the ADR data loader. The same back-end configuration file is used by the dataset registration utility program. In a back-end configuration file, each host has an entry which contains the following information.

1. a hostname,
2. a unique *logical processor id*,
3. a set of *local disk specification records*, where each corresponds to a local disk,
4. a set of *remote disk specification records*, where each corresponds to a remote physical disk that is mounted onto the local file system.

A local disk specification record consists of the following fields:

1. the label "l",
2. a physical disk id that is unique in the entire back-end configuration file,
3. a list of logical disk id ranges, separated by blanks, assigned to this physical disk, where each range is in one of the following format (n and m are two non-negative integers):
   (a) n : this corresponds to a single logical disk id,
   (b) n-m : this corresponds to a set of logical disk id's from n up to m ($n \leq m$),
4. a terminating value "-1".

A physical disk id and a logical disk id can only appear once in exactly one local disk specification record throughout the entire back-end configuration file. A remote disk specification record consists of the following fields:

```
# back-end configuration file

host1                # hostname
0                    # logical processor id
l 0     0 -1         # physical disk 0 is a local disk, assigned logical disk 0
r       1-2 -1       # host1 can only access remote physical disk 1-2
===

host2
1                    # logical processor id
l 1     1-2 -1       # physical disk 1 is a local disk, assigned logical disk 1,2
l 2     3 -1         # physical disk 2 is a local disk, assigned logical disk 3
r       0 -1         # host2 can access remote physical disk 0
===
```

Figure 26: An ADR back-end configuration file example.

1. the label "r",
2. a list of logical disk id ranges, separated by blanks, assigned to this physical disk, where each range is in one of the following format (n and m are two non-negative integers):
   (a) n : this corresponds to a single physical disk id,
   (b) n-m : this corresponds to a set of physical disk id's from n up to m (must have $n \leq m$),
3. a terminating value "-1".

A physical disk id may appear in multiple remote disk specification records in the back-end configuration file. Each entry in the back-end configuration file is terminated by an end-of-entry string, "===".

For example, the back-end configuration file in Figure 26 specifies that there are two ADR back-end nodes, where host1 has one local disk with physical disk id 0, and host2 has two local disks with physical disk id 1 and 2. The local disk of host1 is cross-mounted on host2, and the two local disks of host2 are also cross-mounted on host1. For better I/O bandwidth, the parallel ADR back-end by default assigns to each back-end node its local disks. As will be described later, dataset files are assigned to logical disks. The default assignment therefore assigns a dataset file to the back-end node, the local physical disk of which contains the corresponding logical disk. For example, given the back-end configuration file in Figure 26, all dataset files assigned to the logical disk id 1 would be assigned to host2. As is described in Section 6.7, this default assignment can be overridden at run-time by a command-line parameter to the parallel ADR back-end program.

Registration of datasets is done with an ADR utility program, called `register-datasets`.

```
% register-datasets <dataset registration file> \
            <back-end configuration file> <constructor catalog file> \
            <dataset information file> \
            <dataset catalog file> <index catalog file>
```

This utility program reads a *dataset registration file*, consults the back-end configuration file described earlier and the constructor catalog file, which is generated by another utility program, called `register-constructors`, as was described in Section 6.4. `register-datasets` generates the following three output files.

- a *dataset information file*, which will be used by the ADR front-end to answer dataset inquiries from application front-ends,
- a *dataset catalog*, which contains information about all datasets registered with ADR, and
- an *index catalog*, which contains information about all indexes for datasets registered with ADR.

Each dataset has an entry in the dataset registration file, and each entry has the following format.

1. a unique dataset name,
2. the name of the dataset constructor function for this dataset,
3. an optional filename for the blob object of the dataset (an empty line means no blob object),
4. an optional one-line description for the dataset (an empty line means no description),
5. a list of dataset filenames and the logical disk id's the files are assigned to, with each dataset filename preceded by the label "d",
6. an optional list of auxiliary dataset filenames, with each filename preceded by the label "a",
7. a list of index records, with each record in the following format:
   (a) a unique index name,
   (b) the name of the index constructor function for this index,
   (c) an optional one-line description for the index (an empty line means no description)
   (d) a list of index file records, each of which is in the following format.
      i. the label "i"
      ii. the index filename
      iii. a list of dataset file id ranges, with each range in one of the following formats.
         - n, which corresponds to a single dataset file id,
         - n-m, which corresponds to a set of dataset file id's from n upto m ($n \leq m$)

   (e) the number "-1" to terminate the index file record
8. the string "===" to terminate the entry.

Figure 27 shows an example of a dataset registration file. Dataset names are expected to be unique. The utility program `register-datasets` fails if it sees dataset names that are not unique.

For a dataset with multiple dataset iterators, the id of a dataset iterator is implicitly defined by the order it appears in the dataset entry, with the first iterator starting with id equal to zero. Similarly, each dataset file is also associated with a dataset file id, defined by the order the dataset file appears in the dataset entry. The first dataset file has its dataset file id equal to zero. The

dataset file id's are used by the index records to associate the index files with the dataset files. If an index file contains information about one or more clusters for a given dataset file, then this index file is associated with that dataset file, whose dataset file id should appear in the index record that corresponds to the given index file. Therefore, for a given dataset, each of its dataset file id's should be associated with at least one index file. For example, given the example in Figure 27, the index file `datasets/dataset-1-index.1` keeps the information about all clusters stored in the dataset file `datasets/dataset-1-file.1`, and `datasets/dataset-1-file.2`. Naturally, this association also implies that an index file should be assigned to all the logical disk id's that its associated dataset files are assigned to.

```
# dataset registration file

t2-svm-example:image-dataset-1               # dataset name
t2-svm-example:image-dataset-constructor     # constructor name
image-dataset-thumbnail                      # filename for blob object
description for dataset-1
d datasets/dataset-1-file.0         0        # data file assigned to logical disk 0
d datasets/dataset-1-file.1         1        # data file assigned to logical disk 1
d datasets/dataset-1-file.2         2        # data file assigned to logical disk 2
d datasets/dataset-1-file.3         3        # data file assigned to logical disk 3
d datasets/dataset-1-file.4         0        # data file assigned to logical disk 0
a datasets/dataset-1-aux-file                # auxiliary dataset file for all nodes
t2-svm-example:image-dataset-1-index         # index name
t2-svm-example:image-dataset-index-constructor  # constructor name
description for index-1
i datasets/dataset-1-index.0        0   -1   # index file for dataset file 0
i datasets/dataset-1-index.1        1-2 -1   # index file for dataset file 1-2
i datasets/dataset-1-index.2        3   -1   # index file for dataset file 3
===                                          # end-of-record

# entry 2 starts here ...
```

Figure 27: An ADR dataset registration file example.

A dataset catalog file maintains information about datasets registered through the dataset registration process. A dataset catalog has a header to summarize information obtained from the back-end configuration file. After the header, it keeps an entry for every dataset available in the dataset registration file. Figure 29 shows an example of a dataset catalog.

The dataset catalog header stores the number of logical processor id's and the number of logical disk id's, obtained from the back-end configuration file. It then keeps a list of entries, one per hostname listed in the back-end configuration file, with each entry in the following format.

1. a hostname,
2. the assigned logical processor id,
3. a list of assigned logical disk id's.

Recall that by default, a back-end node is responsible for accessing all dataset files that are assigned to the logical disk id's assigned to the back-end node.

```
# dataset information file

1                                       # total number of datasets

t2-svm-example:image-dataset-1          # dataset name
0                                       # dataset id
description for dataset-1               # dataset description
image-dataset-thumbnail                 # blob filename
1                                       # number of iterators
iterator-1                              # name for iterator 0
1                                       # number of indexes
t2-svm-example:image-dataset-1-index    # name for index 0
0                                       # index id
description for index-1                 # index description
```

Figure 28: An ADR dataset information file example.

After the header, the dataset catalog keeps the number of datasets registered with ADR, and has an entry for each of those datasets, with each entry in the following format.

1. a unique dataset id,
2. the id of its constructor function, obtained by looking up the constructor catalog,
3. the number of dataset files for this dataset,
4. a list of <dataset file, assigned logical disk id> pairs, with one pair per dataset file,
5. the number of auxiliary dataset file for this dataset,
6. a list of paths for the auxiliary dataset files.

An index catalog file maintains information about indexes registered through the dataset registration process. Figure 30 shows an example of an index catalog. Each entry in the catalog is in the following format.

1. a unique index id,
2. the id of its constructor function, obtained by looking up the constructor catalog,
3. the number of index files for this index,
4. a list of index file records, with each of which in the following format.
   (a) a path to the index file,
   (b) the number of dataset file id ranges,
   (c) a list of dataset file id ranges, where each can be in one of the following formats:
      - n: represent a single dataset file id,
      - n-m: represent a set of dataset file id's from n to m (must have $n \leq m$).

```
# dataset catalog

# header
2       4                           # number of logical proc id's and logical disk ids
host1   0                           # host1 is assigned logical proc id 0
        0 -1                        # and logical disk id 0
host2   1                           # host2 is assigned logical proc id 1
        1 2 3 -1                    # and logical disk id 1,2,3

# dataset entry 1
1                                   # number of datasets in the catalog
0                                   # dataset id 0
0                                   # uses dataset constructor function id 0
5                                   # number of dataset files
datasets/dataset-1-file.0       0   # dataset file 0 assigned to logical disk id
0
datasets/dataset-1-file.1       1   # dataset file 1 assigned to logical disk id
1
datasets/dataset-1-file.2       2   # dataset file 2 assigned to logical disk id
2
datasets/dataset-1-file.3       3   # dataset file 3 assigned to logical disk id
3
datasets/dataset-1-file.4       0   # dataset file 4 assigned to logical disk id
0
1                                   # number of auxiliary dataset files
datasets/dataset-1-aux-file         # aux dataset file 0
```

Figure 29: An ADR dataset catalog file example.

Recall that the dataset registration file records the association between index files and the dataset files. The dataset file id ranges in the index file records also keep the specified association.

6.7 Running ADR Front-end and Back-end

Running the ADR system is achieved by running first the ADR front-end program, `t2fe`, and then the parallel ADR back-end program, `pbe`.

The ADR front-end `t2fe` is started with the following command-line arguments.

- -d <dataset information file>, to specify the path to the dataset information file generated by the utility program `register-datasets`, as described in Section 6.6,
- -f <constructor catalog file>, to specify the path to the constructor catalog file generated by the utility program `register-constructors`, as described in Section 6.4,
- -b <back-end port number>, to specify the port number that the parallel ADR back-end nodes should connect to,

```
# index catalog

1                               # number of entries in this catalog

0                               # index id 0
0                               # uses index constructor function id 1
3                               # number of index files
datasets/dataset-1-index.0      # index file 0
        1 0                     # associated with one dataset file id range (0)
datasets/dataset-1-index.1      # index file 1
        1 1-2                   # associated with one dataset file id range (1-2)
datasets/dataset-1-index.2      # index file 2
        1 3                     # associated with one dataset file id range (3)
```

Figure 30: An ADR index catalog file example.

- -p <application front-end port number>, to specify the port number that the application front-ends should connect to.

In addition, `t2fe` also allows the following optional command-line arguments, often used for debugging purposes.

- -h, to print out the list of valid command-line arguments,
- -a <maximum number of simultaneously connected application front-ends>, to specify the maximum number of application front-ends that can connect to the ADR front-end at any given time; application front-ends that attempt to connect after the given maximum number of application front-ends have been reached will get a denial message indicating that the ADR front-end is busy,
- -c <number of application front-ends to serve>, to make `t2fe` terminates as soon as it becomes idle after the given number of application front-ends have connected and disconnected,
- -t <number of seconds to wait for ADR back-end nodes>, to specify the number of seconds to wait before any ADR back-end node connects to `t2fe`; if no ADR back-end node connects to this ADR front-end within the given time, the ADR front-end runs in test mode, in which case the ADR front-end only honors all inquiries from the application front-ends for datasets and constructor functions, but discard all queries that it receives,
- -q <query batch file>, to start the ADR front-end in debugging mode, in which case the ADR front-end does not accept connections from any application front-end, but simply reads an ADR query from the given file and forwards it to the parallel ADR back-end nodes some fixed number of times, which is specified by the "-n" option described below,
- -n <number of times>, which specify in the debugging mode, the number of times the ADR query that the ADR front-end repeatedly sends the query it reads from the file specified by the "-q" option to the parallel ADR back-end.

Note that the "-c" option effectively specifies the minimum number of application front-ends that the ADR front-end should serve before exiting. Even after the specified number of application front-ends have connected to and disconnected from the ADR front-end, the ADR front-end would not exit and would still accept new connections from application front-ends as long as there is always at least one application front-end connected or the ADR back-end nodes have not processed all the queries submitted to the ADR front-end. To keep the ADR front-end running without exiting, use "-c 0".

Figure 31 summarizes the command-line arguments for the ADR front-end program. Once the ADR front-end runs, it waits for the ADR back-end nodes to connect. After the back-end nodes connect or time expires if the option "-t" was specified, the ADR front-end waits for the application front-ends to connect.

```
t2fe -d <dataset information file>
     -f <constructor catalog file>
     -b <back-end port number>
     -p <application front-end port number>
     [-h]
     [-a <maximum number of simultaneously connected application front-ends>]
     [-c <number of application front-ends to serve>]
     [-t <number of seconds to wait for ADR back-end nodes>]
     [-q <query batch file>]
     [-n <number of times to repeatedly send a query to ADR back-end>]
```

Figure 31: Summary of the command-line arguments for the ADR front-end program. Arguments enclosed in brackets "[]" are optional.

The parallel ADR back-end, `pbe`, can be started using whatever utility the target parallel machine provides for loading and executing MPI programs. For example, the IBM SP uses its Parallel Operating Environment (POE) to load and execute MPI programs compiled by its mpcc/mpCC compilers. However, MPICH uses a utility program, `mpirun`, to load and execute MPI programs compiled on the platforms that MPICH supports.

When the parallel ADR back-end, `pbe`, is started, it requires the following command-line arguments.

- -d <dataset catalog file>, to specify the path to the dataset catalog file generated by the utility program `register-datasets`, as described in Section 6.6,
- -i <index catalog file>, to specify the path to the index catalog file generated by the utility program `register-datasets`, as described in Section 6.6,
- -m <memory size (bytes) for accumulator>, to specify the amount of memory that should be used on each back-end node for accumulators,
- -f <ADR front-end hostname> <ADR front-end port number>, to specify the host and the port number that each back-end node should connect to; note that this port number should be equal to the port number specified by the "-b" option when the ADR front-end is started.

In addition, there are a number of options that are mainly used for debugging purpose.

- -h, to print out the list of valid command-line arguments,

- -q <query batch file>, to start the parallel ADR back-end in debugging mode, in which case the back-end nodes do not connect to the ADR front-end, but simply read a query from the given file, process the query and exit,
- -o <output file name>, to force the ADR back-end to ignore the output-handling methods specified in the queries and write the output generated from processing ADR queries to files using the given file name as a prefix; the complete file name has the following format:

```
<output file name>-<output tile number>.<back-end node MPI rank>
```

The ADR back-end program also has a couple of options to override the assignment between dataset files and the ADR back-end nodes. Recall that by default, each back-end node is assigned a logical processor id and a number of logical disk id's, as specified by the back-end configuration file. This effectively defines a mapping from the logical processor id's to the logical disk id's. On the other hand, the dataset registration file defines a mapping from the dataset files to the logical disk id's. This two mapping therefore gives us the default assignment between dataset files and back-end nodes. Specifically, each back-end node is responsible for accessing the dataset files that are assigned to the logical disk id's assigned to that back-end node. This assignment is stored in the dataset catalog. At startup time, the ADR back-end program allows users to override the default assignment between back-end nodes and logical disk id's through one of the following command-line arguments.

- -r, this command-line argument ignores the logical processor id's and simply assigns logical disk id's to back-end nodes in a round-robin order, based on the MPI ranks of the back-end nodes; that is, logical disk id 0 is assigned to the back-end node with MPI rank equal to 0, logical disk id 1 is assigned to the back-end node with MPI rank equal to 1, and so on.
- -a <disk-assignment file>, this command-line argument assigns logical disk id's to back-end nodes according to the mapping specified by the disk-assignment file.

The disk-assignment file allows more flexible assignment between the logical disk id's and the back-end nodes. Users can explicitly assign logical disk id's to back-end nodes, based on the hostnames or the MPI ranks of the back-end nodes. The dataset files are then assigned to the back-end nodes in an obvious way. Alternatively, users can explicitly assign logical processor id's to back-end nodes, based on the hostnames or the MPI ranks of the back-end nodes, and then reuse the mapping between the logical processor id's and the logical disk id's stored in the dataset catalog to assign dataset files to the back-end nodes. This allows users to easily substitute a host listed in the back-end registration file with another host.

Figure 32 and 33 show two disk-assignment files that assign logical disk id's to back-end nodes according to hostnames or MPI ranks, respectively. The two-letter key words at the beginning of those two files specify whether the mapping is from hostname ("h") to logical disk id's ("d") or from MPI ranks ("r") to logical disk id's ("d"). Each back-end node would scan through the given disk-assignment file and look for the assigned logical disk id's assigned either to its hostname (Figure 32) or to its MPI ranks ((Figure 33). The value "-1" on each line marks the end of the list of logical disk id ranges.

Figure 34 and 35 show two disk-assignment files that assign logical processor id's to back-end nodes according to hostnames or MPI ranks, respectively. The two-letter key words at the beginning of those two files specify whether the mapping is from hostname ("h") to logical processor id's ("p")

```
# disk assignment file
hd                      # map hostnames to logical disk ids
4                       # number of valid logical disk ids starting from 0

host3    0-1 -1         # host3 is assigned logical disk id 0 and 1
host4    2-3 -1         # host4 is assigned logical disk id 2 and 3
```

Figure 32: An example of a disk-assignment file that assigns logical disk id's according to back-end hostnames.

```
# disk assignment file
rd                      # map processor rank to logical disk ids
4                       # number of valid logical disk ids starting from 0

0        0-2 -1         # MPI rank 0 is assigned logical disk id 0, 1, 2
1        3 -1           # MPI rank 1 is assigned logical disk id 3
```

Figure 33: An example of a disk-assignment file that assigns logical disk id's according to MPI ranks of back-end nodes.

or from MPI ranks ("r") to logical processor id's ("p"). Each back-end node would scan through the given disk-assignment file and look for the assigned logical processor id's assigned either to its hostname (Figure 32) or to its MPI ranks ((Figure 33). It then uses the dataset catalog to find out the logical disk id's assigned to the set of logical processor id's it assumes. The value "-1" on each line marks the end of the list of logical processor id ranges. Figure 36 summarizes the command-line arguments for the ADR back-end program.

```
# disk assignment file
hp                      # map hostnames to logical proc ids
2                       # number of valid logical processor ids starting from 0

host3    0 -1           # host3 is assigned logical processor id 0
host4    1 -1           # host4 is assigned logical processor id 1
```

Figure 34: An example of a disk-assignment file that assigns logical processor id's according to back-end hostnames.

```
# disk assignment file
rp                      # map processor rank to logical proc ids
2                       # number of valid logical processor ids starting from 0

0     1 -1              # MPI rank 0 is assigned logical processor id 1
1     0 -1              # MPI rank 1 is assigned logical processor id 0
```

Figure 35: An example of a disk-assignment file that assigns logical processor id's to back-end MPI ranks.

```
pbe  -d <dataset catalog file>
     -i <index catalog file>
     -m <memory size (bytes) per back-end node for accumulator>
     -f <ADR front-end hostname> <ADR front-end port number>
     [-h]
     [-q <query batch file>]
     [-o <output file name>]
     [-r]
     [-a <disk-assignment file>]
```

Figure 36: Summary of the command-line arguments for the ADR back-end program. Arguments enclosed in brackets "[]" are optional.

A ADR Utility Classes

This section lists the methods provided by ADR utility classes. These classes are intented to be used in customization of ADR services.

- `T2_AccMsgBuffer`: an buffer for communicating the accumulator elements during the global combine phase; it is typedef'ed to class `T2_UsrArg`, and is always presented to the user either as class `T2_AccMsgBufferWriter`, which is typedef'ed to class `T2_UsrArgWriter`, and class `T2_AccMsgBufferReader`, which is typedef'ed to class `T2_UsrArgReader`.
- `T2_Array<`*type*`>`: a template for an array of object of class *type*; the class has three access methods:
  - `u_int getNumberElements()`: returns the size of the array in terms of the number of elements
  - `const type* getElements() const`: returns the pointer to the actual element array
  - `type* getElements()`: returns the pointer to the actual element array
  - `const` *type*`& operator[] (u_int i)` const: returns an immutable reference to the i-th element
  - *type*`& operator[] (u_int i)`: returns a mutable reference to the i-th element
  - `void resize(u_int newsz)`: resize the array and keep the old elements whenever possible; if the new array is shorter, elements from the old array are truncated
  - `void clear(u_int newsz)`: resize the array and discard all elements
- `T2_BlockRequest`: an object to store a file id, an offset and a block size; it has the following methods:
  - `T2_DSFileID getFileID() const`: returns the file id of a chunk request (which is not the file descriptor)
  - `T2_DSFileID& getFileID()`: returns a reference to the file id of a chunk request (which is not the file descriptor)
  - `off_t getOffset() const`: returns the offset into the data file for the chunk request
  - `off_t& getOffset()`: returns a reference to the offset into the data file for the chunk request
  - `size_t getNumberBytes() const`: returns the size of the chunk in terms of bytes
  - `size_t& getNumberBytes()`: returns a reference to the size of the chunk in terms of bytes
- `T2_Box`: a hyper-rectangle in some multi-dimensional attribute space (*e.g.*, (10,20,50)-(40,100,60)); it consists of two points, a *low* point and a *high* point, which specify the lower and upper bounds of the hyper-rectangle respectively; it has the following methods
  - `T2_Box(u_int ndim=0)`: a constructor that specifies the number of dimension of the box
  - `T2_Box(const T2_Point& low, const T2_Point& high)`: a constructor that specifies the lower and upper bounds of the hyper-box; the two points must have the same number of dimensions

  - `T2_Box(const T2_Box& b)`: a copy constructor
  - `T2_Box(const T2_Point& p)`: to construct a box with both its lower and upper bounds set to the same point, `p`
  - `u_int getNumberDimensions() const`: returns the number of dimensions of the box (*i.e.*, the cardinality of the coordinates)
  - `void setNumberDimensions(u_int d)`: set the number of dimensions
  - `const T2_Point& getLow() const`: get the low point of the box
  - `const T2_Point& getHigh() const`: get the high point of the box
  - `T2_Point& getLow()`: get a reference to the low point of the box
  - `T2_Point& getHigh()`: get a reference to the high point of the box
  - `bool contains(const T2_Box& box) const`: returns true if the argument is wholly contained by the owner box of the method
  - a set of operators between two boxes, which only work if the two boxes have the same number of dimensions (the assignment operator also works if the box on the left-hand-side has its number of dimensions equal to zero)
    * `T2_Box& operator = (const T2_Point &p)`: assignment from another point
    * `T2_Box& operator = (const T2_Box &b)`: assignment from another box
    * `bool operator == (const T2_Box& p) const`: equality test
    * `bool operator != (const T2_Box& p) const`: inequality test
    * `bool operator ^ (const T2_Point& p) const`: returns true if the argument point is contained within the box
    * `bool operator ^ (const T2_Box& p) const`: intersection test

- `T2_Cluster`: a set of `(void*, size_t)` pairs, each of which represents a pointer to a contiguous data block and its size; it has the following methods:
  - `u_int getNumberBlocks() const`: returns the number of blocks retrieved from disks
  - `const char* getDataPointer(u_int i) const`: returns the pointer to the i-th block
  - `size_t getDataSize(u_int i) const`: returns the number of bytes of the i-th block

- `T2_ClusterAuxInfoWriter`: allows user-defined meta-data for a chunk, retrieved during index search, to be written into a buffer. It is a subclass of `T2_UsrArgWriter`, and implements the same methods. The definition of `T2_UsrArgWriter` is given in this section.

- `T2_ClusterAuxInfoRead` is used to read the user-defined meta-data for a chunk from the buffer encapsulated by `T2_ClusterAuxInfoWriter`. It is a subclass of `T2_UsrArgReader`, and implements the same methods. The definition of `T2_UsrArgReader` is given in this section.

- `T2_ClusterInfoList`: contains information for a list of input data chunks of all datasets accessed by a given query, one entry per data chunk; it has the following methods
  - `u_int getNumberDatasets()`: which returns the number of datasets accessed by the query
  - `T2_DSID getDatasetID(u_int d)`: which returns the dataset ID of the `d`-th dataset specified in the query

- `u_int getNumberClusters(u_int d)`: which returns the number of clusters to be read by the `d`-th dataset specified in the query; note that this is *not* the same as the dataset ID, which is returned by method `getDatasetID`
- `const T2_Box& getBoundingBox(u_int d, u_int i) const`: which returns the bounding box of the `i`-th input data chunk of the `d`-th dataset; note that `i` is just used to specify the `i`-th input data chunk on the list among all the data chunks of a particular dataset and the execution does not promise to read the data chunks by the order they are listed on the list
- `bool getMetaData(u_int d, u_int i, T2_UsrArgReader& reader) const`: which returns the per-cluster meta-data provided by the index when it is looked up; note that the caller must provide an object of `T2_UsrArgReader` as the third argument, which could be used to read the meta-data after the method returns; returns `true` if `d` and `i` is within the appropriate range, `false` otherwise
- `bool getClusterInfo(u_int d, u_int i, T2_Box& box, T2_UsrArgReader& reader)`: which gets both the bounding box and the meta data of a particular data chunk with one function call

- `T2_FEDatasetEntry`: a class to hold the information for a dataset as a result that the ADR front-end returns in response to a dataset inquiry
  - `T2_DSID getDatasetID() const`: returns the unique dataset id
  - `const char* getDatasetName() const`: returns the dataset's unique name
  - `const char* getDatasetDescription() const`: returns the dataset one-line description, if any
  - `size_t getBlobDataSize() const`: returns the size of the dataset blob object, if any
  - `const char* getBlobObject() const`: returns the content of the dataset blob object, if any
  - u_int getNumberIterators() const: returns the number of iterators for this dataset
  - `const char* getIteratorName(u_int i) const`: returns the name of the i-th iterator
  - u_int getNumberIndexes() const: returns the number of indexes for this dataset
  - `const T2_IndexInfoEntry& getIndex(u_int i) const`: returns the information of the i-th index

- `T2_FEDatasetInquiryResults`: a class to hold the results for all datasets returned by the T2 front-end in response to a dataset inquiry; this is effectively an array of `T2_FEDatasetEntry` objects
  - `T2_FEInquiry::inquiry_result_tag getStatus() const`: return the status of the inquiry, which could be one of the following constants:
    - `T2_FEInquiry::results_ok`: the results are complete and correct
    - `T2_FEInquiry::bad_socket_error`: the results are not valid due to some socket error occurs during the inquiry
    - `T2_FEInquiry::invalid_inquiry`: the inquiry was invalid
    - `T2_FEInquiry::t2_frontend_not_ready`: the ADR front-end is not ready to answer inquiries for datasets

- `u_int getNumberEntries() const`: returns the number of `T2_FEDatasetEntry` entries in this object
- `const T2_FEDatasetEntry& operator [] (u_int i) const`: returns a reference to the i-th entry in this object

- `T2_FEFunctionEntry`: a class to hold the information for a constructor function as a result that the ADR front-end returns in response to a constructor function inquiry
  - `T2_UDFType getFunctionType() const`: return the function type, which could be one of the following constants:
    - `T2_UDF_Unknown`: an unknown/invalid type
    - `T2_UDF_AccMeta`: a constructor function for an accumulator meta-data object
    - `T2_UDF_Aggregation`: a constructor function for an aggregation function object
    - `T2_UDF_Projection`: a constructor function for a projection function object
  - `u_int getFunctionID() const`: returns the constructor function id
  - `const char* getFunctionName() const`: returns the constructor function name
  - `const char* getFunctionDescription() const`: returns a one-line description for the constructor function, if any

- `T2_FEFunctionInquiryResults`: a class to hold the results for all constructor functions returned by the T2 front-end in response to a constructor function inquiry; this is effectively an array of `T2_FEFunctionEntry` objects
  - `T2_FEInquiry::inquiry_result_tag getStatus() const`: return the function type, which could be one of the following constants:
    - `T2_UDF_Unknown`: an unknown/invalid type
    - `T2_UDF_AccMeta`: a constructor function for an accumulator meta-data object
    - `T2_UDF_Aggregation`: a constructor function for an aggregation function object
    - `T2_UDF_Projection`: a constructor function for a projection function object
  - `u_int getNumberEntries() const`: return the number of `T2_FEFunctionEntry` objets
  - `const T2_FEFunctionEntry& operator [] (u_int i) const`: return a reference to the i-th entry in this object

- `T2_IndexInfoEntry`: a class to hold the information for a user-defined index
  - `T2_IndexID getIndexID() const`: returns the unique index id
  - `const char* getIndexName() const`: returns the index name
  - `const char* getIndexDescription() const`: returns the index one-line description, if any

- `T2_OutputBuffer`: a buffer that the final output uses for flattening out its data into contiguous space when necessary; it provides the same set of methods provided by `T2_UsrArgWriter` with the addition of the following method.
  - `const char* getCurrentPosition() const`: returns the position currently pointed to by the buffer pointer

- `T2_OutputDataPtrList`: an ordered list of pairs of `char*` pointers and data sizes; it has the following methods
  - `void insertDataPointer(const char* p, size_t s)`: which allows pointer-size pairs to be appended to the list
- `T2_Point`: a point in some multi-dimensional attribute space (*e.g.*, (3, 50, 29)); currently each coordinate of the point is a `float`; has the following methods,
  - `T2_Point(u_int ndim=0)`: a constructor that specifies the number of dimension of the point
  - `T2_Point(u_int ndim, const float* c)`: a constructor that takes an array of `ndim` floating point numbers to initialize the coordinates of the point
  - `T2_Point(const T2_Array<float>&)`: a constructor that takes an array of floating point numbers to initialize the coordinates of the point
  - `u_int getNumberDimensions() const`: returns the number of dimensions of the point (*i.e.*, the cardinality of the coordinates)
  - `void setNumberDimensions(u_int d)`: set the number of dimensions
  - `const float& operator[](const u_int i) const`: returns the i-th coordinate
  - `float& operator[](const u_int i)`: returns a reference to the i-th coordinate
  - `T2_Point& offset(const T2_Point& p)`: updates the point by adding the argument p to the point coordinate-wise
  - a set of operators between two points, which only work if the two points have the same number of dimensions (the assignment operator also works if the point on the left-hand-side has its number of dimensions equal to zero)
    - `T2_Point& operator = (const T2_Point &p)`: assignment from another point
    - `bool operator == (const T2_Point& p) const`: equality test
    - `bool operator != (const T2_Point& p) const`: inequality test
    - `bool operator ^ (const T2_Point& p) const`: intersection test
- `T2_QueryInfo`: a class that provides information about the back-end system (see the description about `T2_System`) and the query being processed
  - `u_int getNumberNodes() const`: returns the number of back-end nodes,
  - `T2_ProcID getMyNode() const`: returns the local processor id
  - `u_int getBackEndEndian() const`: returns either the constant `T2_BigEndian` or the constant `T2_LittleEndian` to show which endian is used on the back-end nodes
  - `u_int getExpectedQueryOutputEndian() const`: returns either the constant `T2_BigEndian` or the constant `T2_LittleEndian` to show which endian should be used for the query output
  - `T2_Iteration getNumberTiles() const`: return the number of output tiles
  - `T2_Iteration getCurrentTileNumber() const`: return the number of the tile that the ADR back-end is currently working on

- `T2_Region`: a list of `T2_Box`'es in some multi-dimensional attribute space (*e.g.*, {(1,5)-(2,7), (10,4)-(11,5)} represents the following points: (1,5), (1,6), (1,7), (2,5), (2,6), (2,7), (10,4), (10,5), (11,4), (11,5))
  - `T2_Region(u_int ndim=0, u_int maxnboxes=0)`: a constructor that specifies the number of dimensions (`ndim`) and the maximum number of boxes that can be stored in the region (`maxnboxes`)
  - `T2_Region(const u_int n, const T2_Box* b)`: a constructor that initializes the region with an array of `n` `T2_Box`'s pointed to by the pointer `b`
  - `T2_Region(const T2_Region& r)`: a copy constructor
  - `T2_Box(const T2_Box& b)`: to construct a region as a set of a single box `b`
  - `T2_Box(const T2_Point& p)`: to construct a region as a set of a single box whose lower and upper bounds are both set to the same point, `p`
  - `u_int getNumberDimensions() const`: returns the number of dimensions of the point (*i.e.*, the cardinality of the coordinates)
  - `void setNumberDimensions(u_int d)`: set the number of dimensions
  - `u_int getNumberBoxes() const`: returns the number of `T2_Box`'es in the region
  - `u_int getMaxSetSize() const`: returns the maximum number of `T2_Box`'es that can be stored with the region
  - `void growMaxSetSize(u_int l)`: grow the maximum number of `T2_Box`'es that can be stored with the region to the give argument
  - `T2_Region& operator += (const T2_Box& box)`: add a box to the region
  - `T2_Region& operator << (const T2_Box& box)`: add a box to the region
  - `const T2_Box& operator[](const u_int i) const`: returns the i-th box
  - `T2_Box& operator[](const u_int i)`: returns a reference to the i-th box
  - `bool contains(const T2_Box& box) const`: returns true if the argument is wholly contained in the region
  - `bool contains(const T2_Region& reg) const`: returns true if the argument is wholly contained in the region
  - `void getMBR(T2_Box& box) const`: returns a bounding box of the entire region
  - a set of overloaded assignment operators
    - `T2_Region& operator = (const T2_Point& pt)`: make the region a region that contains only one point, which is the given argument
    - `T2_Region& operator = (const T2_Box& box)`: make the region a region that contains only one box, which is the given argument
    - `T2_Region& operator = (const T2_Region &r)`: assignment from another region
  - a set of operators between two regions, which only work if the two points have the same number of dimensions (the assignment operator also works if the region on the left-hand-side has its number of dimensions equal to zero)
    - `bool operator == (const T2_Region& p) const`: equality test
    - `bool operator != (const T2_Region& p) const`: inequality test
    - `bool operator ^ (const T2_Box& b) const`: intersection test

- `T2_System`: an object that provides system information such as the number of processors and the local processor id; has the following methods,
  - `u_int getNumberNodes() const`: returns the number of back-end nodes,
  - `T2_ProcID getMyNode() const`: returns the local processor id
  - `u_int getBackEndEndian() const`: returns either the constant `T2_BigEndian` or the constant `T2_LittleEndian` to show which endian is used on the back-end nodes
- `T2_UsrArg`: a class used to represent application-dependent arguments; the callee that receives this class is responsible for parsing the arguments correctly; this class actually is presented to the users in two forms, a writer form (`T2_UsrArgWriter`), which allows data to be written into the buffer, and a reader form (`T2_UsrArgReader`), which allows data written into the buffer earlier to be read from the buffer;
  The `T2_UsrArgWriter` class has the following methods:
  - `bool isFull() const`: returns true if the buffer is already full, false otherwise
  - `size_t getBufferSize() const`: returns the buffer size in bytes
  - `size_t getNumberBytesWritten() const`: returns the number bytes that has been written into the buffer
  - `T2_UsrArgWriter& write(const char* c, size_t n)`: allows n bytes pointed by c to be written into the buffer
  - `T2_UsrArgWriter& operator << (`*type* `b)`: allows values of various primitive types (*type* can be char, int, float, double, etc.) to be written into the buffer

  The `T2_UsrArgReader` class provides similar methods:
  - `size_t getDataSize() const`: returns the number of bytes of data that can be read from the buffer
  - `void rewind()`: which allows the buffer pointer to be reset to the beginning of the buffer
  - `T2_UsrArgWriter& read(char* c, size_t n)`: allows `n` bytes to be read from the buffer into the user-buffer
  - `const char* read(size_t n, size_t& actual)`: returns the address that the current pointer points to and advance the current pointer by `n` bytes if possible; if less than `n` bytes are available in the buffer, the current pointer is advanced pass the end of buffer, and `actual` gives the actual number of bytes the current pointer is forwarded; this function is effectively the same as the other `read` method, except it avoids the memory copy made by the other `read` method
  - `T2_UsrArgWriter& operator >> (`*type* `b)`: allows values of various primitive types (*type* can be char, int, float, double, etc.) to be read from the buffer
- `T2_VArray<`*type*`>`: a template for a variable-size array of object of class *type*;
  - `u_int getNumberElements()`: returns the size of the array in terms of the number of elements
  - `const` *type*`& operator[] (u_int i)` const: returns an immutable reference to the i-th element
  - *type*`& operator[] (u_int i)`: returns a mutable reference to the i-th element

– `T2_VArray<`*type*`>& operator << (const` *type*`& e)`: appends an element of *type* to the array

B An Example Application

B.1 The Simplified Virtual Microscope (SVM)

In this section we describe an example application, called *simple virtual microscope* (SVM), to illustrate the customization of ADR services. The SVM is a simplified version of the Virtual Microscope [2, 10], and is based on a subset of the functionality of the Virtual Microscope. The input datasets for the SVM are 2-dimensional PPM images. The output is also a 2-dimensional image, which corresponds to a region of the original input image, viewed at a lower-resolution, specified by a zoom factor. The complete SVM application consists of a SVM client, a SVM front-end, the ADR front-end, and a customized ADR back-end.

The full-resolution images are partitioned into regular rectangular chunks, and stored in the ADR. The index for an image dataset is an array. Each entry of this array corresponds to a rectangular chunk in the original image, and holds the bounding box and location of the chunk in the data files.

The ADR back-end is customized to process queries submitted by the SVM client. A query specifies a rectangular region in the selected image dataset, and a zoom factor to create a lower-resolution image. The customization implements codes to find and access the image chunks that intersect the query region, iterate through the pixels in an image chunk, carry out sub-sampling to create an image at the specified zoom factor, and return the lower-resolution image back to the SVM client.

The ADR front-end maintains information about the datasets available at the ADR back-end, and the registered user-defined functions (e.g., projection, aggregation) that have been implemented as part of the customization. In particular, for each registered SVM image dataset, the ADR front-end stores the user-defined name and the resolution of the image, and a thumbnail for the image. It interacts with one or more SVM front-ends to receive inquiries for dataset and user-defined methods, and to receive ADR queries to be processed by the back-end.

At the start-up, the SVM front-end connects to the ADR front-end and downloads the names, resolutions, and thumbnails of all of the registered SVM datasets, along with the information on the user-defined functions, from the ADR front-end. Each thumbnail image is written to the local disk, to be accessed later for queries from the SVM client. This design has been chosen simply for the ease of implementation. In general, downloading all thumbnail images can be a prohibitively expensive operation. In that case, one could choose to dynamically download the thumbnails as they are needed, and keep a cache of thumbnail images. After downloading meta-data for datasets and user-defined methods, the SVM front-end listens for a connection from a SVM client. Unlike the Virtual Microscope front-end, the SVM front-end currently can only serve one SVM client at a time. Upon accepting a connection, it can receive queries from the client, convert them into ADR queries and submit to the ADR front-end.

The SVM client implements a graphical user interface for a user to select an image dataset, and request a region of the image at a zoom factor. Figure 37 shows the client user interface for the example application. The SVM client first interacts with the SVM front-end to retrieve the names of all SVM datasets registered in ADR. Then, the user can choose a particular image dataset for browsing. The client downloads the size and the thumbnail image of the dataset chosen by the user, and displays the thumbnail image so that user can choose a region in the image. After the user selects a zoom value and a rectangular portion of the image to be displayed, the SVM client sends the selected zoom value and the rectangular region as a request to the SVM front-end, and awaits for the result from the ADR back-end nodes. After processing the query, each ADR back-end node makes a socket connection to the SVM client and sends a portion of the result, i.e., a sub-image of

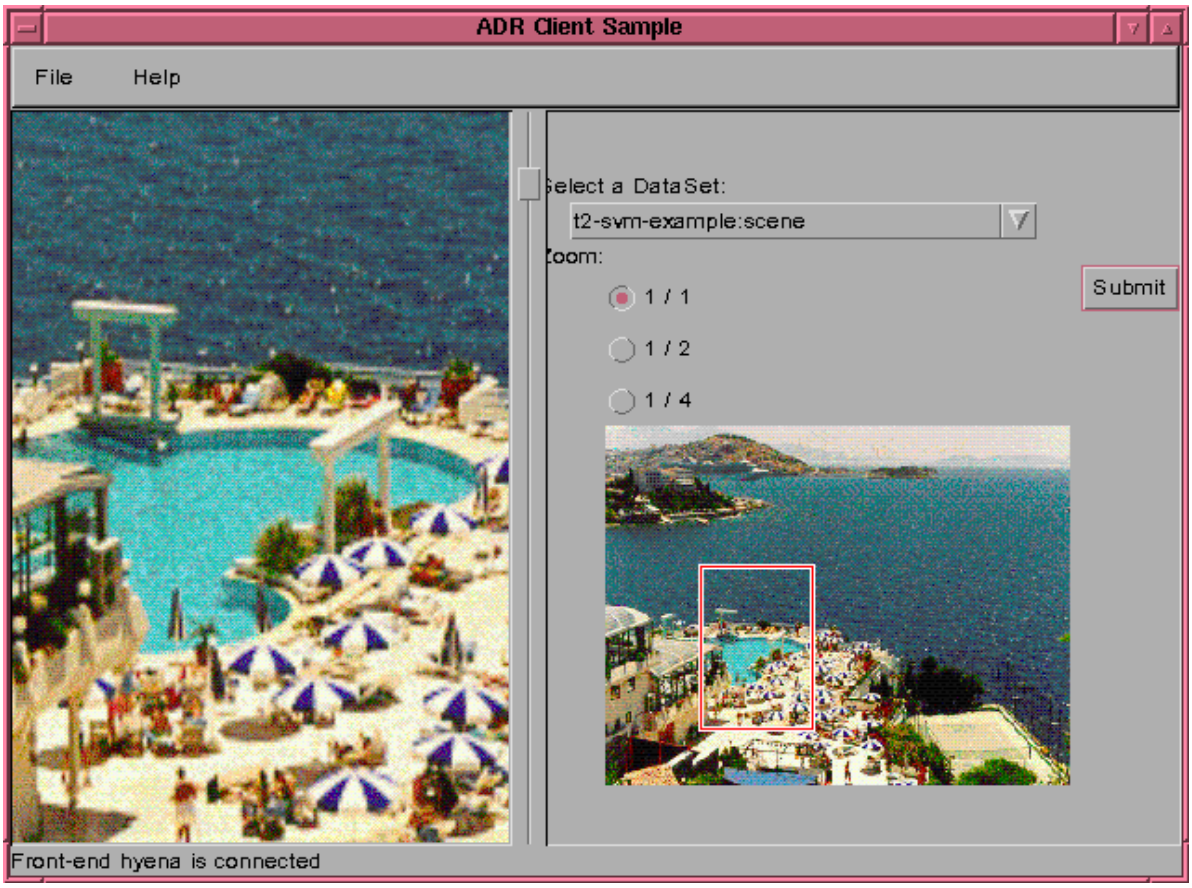

Figure 37: The client interface for the example application.

the entire lower-resolution image back to the SVM client, along with a header information for each sub-image. The SVM client stitches the sub-images back into one output image and displays the image in its display panel.

Section B.2 briefly discusses how ADR is customized for the SVM application. Section B.3 describes how to compile and run the SVM application codes. Section B.4 describes how new datasets (ie images) can be added to the SVM application.

B.2 SVM Customization Codes

In this section, we describe the customization codes for the SVM application. The SVM client is implemented in Java, and is used to select images, the zoom factor, and rectangular regions in an image. Since the client does not use any ADR customization interface, it is not described in this document.

B.2.1 Customization of SVM Front-end

The SVM front-end interacts with both the SVM client and the ADR front-end. In this section, we only focus on its interaction with the ADR front-end.

The SVM front-end interacts with the ADR front-end in the following way.

1. It connects to the ADR front-end to start a session.

2. It inquires the ADR front-end for information about the SVM datasets and functions.

3. It submits ADR queries to the ADR front-end.

4. It disconnects from the ADR front-end to end the session.

Figure 38 shows the SVM front-end implementation– some of the details have been omitted in the figure for the sake of brevity and clarity. The SVM front-end first connects to the ADR front-end using the `T2_FrontEnd::connectT2FrontEndByHostname` method. The two arguments specify the hostname of the machine that the ADR front-end runs on, and the port number that the ADR front-end is listening to. The SVM front-end then calls `svm_inquire_functions` to find out the id of the customized projection function, the id of the customized accumulator meta-data object constructor, and the id of the customized aggregation function for subsampling. It uses the keyword "t2-svm-example", which is the name assigned by the user when functions are registered, to search for the functions. The variables `pid`, `accid`, and `aggid` are used to store projection function id, accumulator meta-data object id, and aggregation function id, respectively. The SVM front-end calls `svm_inquire_datasets` to retrieve the information about the available datasets. The retrieved information is stored in the `svm_ImageInfoCollection` object. The user-defined binary objects for SVM applications are the thumbnails of the images loaded in the ADR. The thumbnail images retrieved as a result of the inquiry are written to the directory specified by the variable `thumbnail_dir` so that the front-end can send the thumbnail image to the client when the client selects the corresponding dataset.

After the information about functions and datasets is retrieved, the front-end can interact with an SVM client. Assume the client selects an image, and requests a rectangular portion of the image, starting at pixel (`x,y`) with `w` pixels wide and `h` pixels high, at a zoom factor `z`, which means every pixel in the output image corresponds to `z`×`z` pixels in the input image. Upon receiving the request, the SVM front-end calls `GenerateQuery` to create an ADR query object, and sends it to the ADR front-end using method `T2_Frontend::submitQBatch`.

We now discuss the three functions that the SVM front-end calls in greater detail. Note that in most of the code segments listed in this section, codes for handling error conditions have been omitted for clarity.

Figure 39 describes the implementation of `svm_inquire_functions`. The variable `func_results` is used to hold the results of the inquiry for user-defined functions. The SVM application assumes that only one SVM projection function, one SVM accumulator meta-data object function, and one SVM aggregation function have been implemented. Furthermore, it is assumed that the names of these functions all start with the word, "t2-svm-example". These assumptions are made to simplify the example code. Therefore, using the `keyword` as a pattern, a call to the method `T2_FrontEnd::inquireFunctionRegExp` can retrieve all functions, whose names contain the keyword "t2-svm-example". The argument `T2_UDF_Unknown` retrieves any type of function registered in the customized ADR, and the argument `func_fields` specifies that only the function id and the function name are requested. The SVM front-end iterates through each entry in the results and sets the function id to proper values, assuming that there is only one user-defined function for each function type.

Figure 40 shows the implementation of `svm_inquire_datasets` function. The SVM application assumes that all datasets names start with the keyword, "t2-svm-example". Therefore, using the `keyword` as a pattern, a call to the method `T2_FrontEnd::inquireDatasetRegExp` retrieves all datasets registered in ADR. The argument `fields` specifies that for each dataset, the dataset id, the dataset name, the dataset description, the dataset blob object, the size of the blob object, and

```
#include "t2_frontend.h"

T2_FrontEnd fe;
if (fe.connectT2FrontEndByHostname(hostname, port) == false)
   // connection fails, handle error;

// inquire of ADR front-end about functions
const char svm_keyword[] = "t2-svm-example";
if (svm_inquire_functions(fe, svm_keyword, pid, accid, aggid)
    == false)
   // inquiry fails, handle error;

// inquire of ADR front-end about datasets (images)
svm_ImageInfoCollection images;
const char thumbnail_dir[] = "thumbnails";   // where to store thumbnails
if (svm_inquire_datasets(fe.svm_keyword, pid, accid, aggid, thumbnail_dir, images)
   == false)
   // inquiry fails, handle error;

// Interact with the SVM client, and create an ADR query object
// for image i, resolution z, and a query region starting from
// (x,y) of w pixels wide and h pixels high ...
GenerateQuery(fe, packno, packtype, x, y, w, h, z, hostname, backendport, images[i]);

fe.disconnectT2FrontEnd();                  // disconnect from ADR front-end
```

Figure 38: Outline of the implementation for the SVM front-end.

```
#include "t2_frontend.h"

svm_inquire_functions(T2_FrontEnd& fe, const char* keyword,
                      u_int& pid, u_int& accid, u_int& aggid)
{
  T2_FEFunctionInquiryResults func_results;         // to hold results
  const int func_fields = T2_FEInquiry::function_id_field |
                          T2_FEInquiry::function_name_field;
  if (fe.inquireFunctionRegExp(keyword, T2_UDF_Unknown, func_fields, func_results)
      == false)
     // inquiry fails, handle error

  for (u_int i=0; i<func_results.getNumberEntries(); i++) {
     T2_FEFunctionEntry& entry = func_results[i];
     switch (entry.getFunctionType()) {
        case T2_UDF_AccMeta:            // accumulator meta object
                        if (accid == 0) accid = entry.getFunctionID();
                        break;
        case T2_UDF_Projection:         // projection function
                        if (pid == 0)   pid = entry.getFunctionID();
                        break;
        case T2_UDF_Aggregation:        // aggregation function
                        if (aggid == 0) aggid = entry.getFunctionID();
                        break;
     }
  }
}
```

Figure 39: The implementation of function `svm_inquire_functions`.

the id of the index is retrieved. These fields are later used to formulate ADR queries. The SVM front-end iterates through each entry in the result of the inquiry, stored in `dataset_results` object. The width and height of the image dataset are assumed to be stored in the dataset description. The blob object, which is assumed to be the thumbnail of the dataset, is written to a file in the specified directory, using the dataset name as the prefix for the thumbnail file.

Figure 41 shows the implementation of `GenerateQuery` that creates an ADR query. This function first creates a query batch object of one single query, which only accesses one input dataset. It then specifies the aggregation function id. The variables `packno` and `packtype` are two integers that the SVM client uses to identify the packets it receives from the ADR back-end. They are sent as the user arguments to the SVM aggregation function, which would pass the two integers to the SVM output object so that they can be embedded in the output result sent from the ADR back-end to the SVM client. The SVM front-end then sets the accumulator meta-data object function id, the dataset id, the dataset iterator id, the dataset index id and the projection function id. It also specifies the rectangular region of interest in the input dataset, and the extent of the output image. Afterwards, the user arguments for the projection function and the accumulator meta-object constructor are specified. The former requires the coordinates of the top-left pixel of the output image and the resolution along both dimensions, while the latter requires the width and height of the output image, along with the number of bytes per output pixel. These values are used by the constructor functions to correctly initialize the projection function and the accumulator meta-data objects. Finally, the SVM front-end specifies that the output should be returned through UNIX sockets to the SVM client, and the hostname and the port number for socket connection.

B.2.2 The SVM Helper Classes

In this section, we briefly describe three classes that will be used by the SVM customization code to be described in the following sections.

- `svm_Image` is a class that stores an image.
- `svm_ImagePixel` is a class that represents a pixel.
- `svm_ImageIterator` is a class that can be used to iterate over a given rectangular region of pixels in an image.

Figure 42 shows the definitions of these helper classes. The `svm_Image` stores an image with the given width and height. The pointer `bytes` points to an array of bytes that correspond to the pixels, stored in the row-major order. The `svm_ImagePixel` represents a pixel with a given number of bytes, which are pointed to by the pointer `bytes`. The `svm_ImageIterator` is an iterator that iterates over a given rectangular region of pixels in an image. The constructor function for `svm_ImageIterator`, as is shown in Figure 42, has eight arguments; `w` and `h` specify the width and height of the given image, `s` specifies the number of bytes per pixel, and `b` points to the pixels, stored in the row-major order. The arguments `x_s`, `y_s`, `x_e` and `y_e` specify a rectangular region of pixels, relative to the top-left corner of the image. In the constructor method of class `svm_ImageIterator`, the class data members `x_start`, `y_start`, `x_end`, `y_end`, `pixelsize`, and `bytes` are initialized accordingly, `x_next` and `y_next` are initialized to the coordinates of the first pixel to return `x_s` and `y_s`, respectively, and the address of the first pixel to be return is computed and assigned to `pixel_next`. When the method `getNextPixel` of class `svm_ImageIterator` is called, it iterates through pixels in the row-major order. The values of `x_next` and `y_next` are stored in the arguments `x_off` and `y_off`, respectively, and the data member `pixel.bytes` is set to the current value of `pixel_next`. The

```
svm_inquire_datasets(T2_FrontEnd& fe, const char* keyword,
                     u_int pid, u_int accid, u_int aggid, const char* thumbnail_dir,
                     svm_ImageInfoCollection& allimages)
{
  T2_FEDatasetInquiryResults dataset_results;
  const int fields = T2_FEInquiry::dataset_datasetid_field |
                     T2_FEInquiry::dataset_datasetname_field |
                     T2_FEInquiry::dataset_datasetdescription_field |
                     T2_FEInquiry::dataset_blobsize_field |
                     T2_FEInquiry::dataset_blobobj_field |
                     T2_FEInquiry::dataset_indexid_field;
  if (fe.inquireDatasetRegExp(keyword, fields, 0, dataset_results) == false)
     // inquiry fails, handle error
  allimages.growSize(dataset_results.getNumberEntries());
  for (u_int i=0; i<dataset_results.getNumberEntries(); i++) {
     svm_ImageInfoEntry*& info_entry = allimages[i];
     info_entry = new svm_ImageInfoEntry;

     info_entry->getDatasetName() = inq_entry.getDatasetName();
     inq_entry.getDatasetName() = NULL;
     info_entry->getDatasetID() = inq_entry.getDatasetID();
     info_entry->getIndexID() = inq_entry.getIndex(0).getIndexID();
     info_entry->getProjectionID() = pid;
     info_entry->getAccMetaID() = accid;
     info_entry->getAggregationID() = aggid;

     // extract width and height from the dataset description
     sscanf(inq_entry.getDatasetDescription(), "%d %d",
            &(info_entry->getImageWidth()), &(info_entry->getImageHeight()));

     // get thumbnail
     info_entry->getThumbnailFileSize() = inq_entry.getBlobDataSize();
     write_thumbnail_file(thumbnail_dir, info_entry->getDatasetName(),
                    inq_entry.getBlobObject(), info_entry->getThumbnailFileSize());
  }
}
```

Figure 40: The implementation of function `svm_inquire_datasets`.

```
GenerateQuery(T2_FrontEnd& fe, int packno, int packtype, int x, int y, int w, int h,
              int z, char *hostname, int backendport, svm_ImageInfoEntry *ie)
{
  T2_QBatch qbatch(1);                    // only one query per batch
  T2_QSpec& qspec = qbatch.getQuerySpec(0);
  qspec.setNumberDatasets(1);             // only access one dataset
  qspec.useBigEndianUserArg();            // use big endian
  T2_UsrArgWriter write_arg;              // user argument writer
  T2_QSpecDataset& dataset_spec = qspec.getDatasetSpec(0);

  qspec.getAggrID() = ie->getAggregationID();
  qspec.getAggrConstructorArg().allocBuffer(sizeof(u_int)*2);
  write_arg.open(qspec.getAggrConstructorArg());
  write_arg << packno << packtype;
  write_arg.close();
  qspec.getAccID() = ie->getAccMetaID();
  qspec.getAccNavigatorID() = 0;          // assuming only 1 acc iterator
  dataset_spec.getDatasetID() = ie->getDatasetID();
  dataset_spec.getIteratorID() = 0;       // assume only 1 dataset iterator
  dataset_spec.getIndexID() = ie->getIndexID();
  dataset_spec.getProjID() = ie->getProjectionID();

  T2_Box& inbox = dataset_spec.getQueryBox();  // set the input query box
  inbox.setNumberDimensions(2);
  inbox.getLow()[0] = x;                  inbox.getLow()[1] = y;
  inbox.getHigh()[0] = x+w;               inbox.getHigh()[1] = y+h;
  T2_Box& output_box = qspec.getOutputBox();   // set the output image extent
  output_box.setNumberDimensions(2);
  output_box.getLow()[0] = 0;             output_box.getLow()[1] = 0;
  output_box.getHigh()[0] = w/z;          output_box.getHigh()[1] = h/z;

  // set projection constructor user arg
  dataset_spec.getProjConstructorArg().allocBuffer(sizeof(u_int)*4);
  write_arg.open(dataset_spec.getProjConstructorArg());
  write_arg << x << y << z << z;
  write_arg.close();
  // set accumulator constructor user arg
  int pixelsize=3;
  qspec.getAccConstructorArg().allocBuffer(sizeof(u_int)*3);
  write_arg.open(qspec.getAccConstructorArg());
  write_arg << w/z << h/z << pixelsize;
  write_arg.close();

  T2_QSpecOutput& outspec = qspec.getOutputSpec();
  outspec.setOutputHandleType(t2_oSocket);
  outspec.setHostName(hostname);          // store the hostname of the client
  outspec.getPortNumber() = backendport; // store the port number client listens to
  outspec.useBigEndianOutput();           // use big endian for output for Java client
  outspec.disableT2Protocol();            // do not use ADR standard protocol
}
```

Figure 41: The implementation of function `GenerateQuery`.

iterator advances to the next pixel by incrementing `x_next` and increasing `pixel_next` by `pixelsize` bytes. However, in the case where the value of `x_next` exceeds the value of `x_end`, `x_next` is reset to `x_start`, `y_next` is incremented, and the pointer `pixel_next` is advanced by `skip_bytes` bytes, which are computed by the constructor of `svm_ImageIterator`.

B.2.3 Customization of Indexing Service

In this section, we describe the SVM customized indexing class derived from the `T2_Index` base class. In the current implementation, each index file is associated with exactly one data file of a dataset. That is, each index file contains the necessary information about all clusters stored in the associated data file. For each cluster, an index file keeps an entry with the following information.

- `x_pos`, `y_pos`: the coordinates of the top-left pixel of the cluster relative to the top-left pixel of the entire input image,
- `width`, `height`: the width and height of the cluster
- pixelsize: the number of bytes per pixel,
- `offset`, `size`, `fid`: these values specify the location of the cluster in the data file,
  - `fid`: specifies the id of the data file of the dataset; recall that this id corresponds to the order that the data file appears in the data registration file (see Section 6.6),
  - `offset`: the offset from the beginning of the data file for the first byte of the cluster,
  - `size`: the size of the cluster in bytes.

The values `x_pos`, `y_pos`, `width` and `height` together define the minimum bounding rectangle of the cluster. For simplicity, all values are stored in the ASCII format, and each entry is stored as a single line.

Figure 43 shows the definition of the customized index class, along with its constructor. When a back-end node is assigned multiple data files of a given dataset, it is also assigned all the index files associated with those data files. The ADR back-end would pass the file descriptors for those index files to the index constructor function.

Figure 44 shows the implementation of the two virtual functions that class `svm_ImageIndex` inherited from the base class, `T2_Index`. When class `svm_ImageIndex` is asked to locate all the clusters intersecting the query box, it needs to go through all entries in all the index files it is assigned to and returns the information of those clusters one-by-one. The class `svm_ImageIndex` therefore uses a counter, `curfp`, to keep track of the current index file it is looking up. The method `svm_ImageIndex::fetchInit` simply initializes `curfp` to 0, and rewinds the first index file to the beginning of the file. The method `svm_ImageIndex::fetch` then simply resumes the search from where it left off before and for every cluster entry that it reads from the current index file, using a function called `my_read`, it computes the minimum bounding rectangle for the query and checks if it intersects with the query box.

B.2.4 Customization of Attribute Space Service

In this section, we describe the implementation of the user-defined projection function for the SVM application. The projection function maps the coordinates of a pixel, defined in the full-resolution input image, to the coordinates of the corresponding pixel in the lower-resolution output image. The coordinate space of the full-resolution image forms the input attribute space, whereas the

```
class svm_Image {
  u_int width, height,             // width and height of an image
        pixelsize;                 // number of bytes per pixel
  u_char *bytes;                   // the byte array of the pixels
};

class svm_ImagePixel {
  u_int pixelsize;                 // number of bytes for this pixel
  u_char* bytes;                   // bytes of this pixel
};

class svm_ImageIterator {         // an iterator for a portion of an image
  u_int x_start, y_start, x_end, y_end,   // the beginning and the end of
                                          // the range of pixels to be
                                          // iterated over
        pixelsize;                 // number of bytes per pixel
  u_char *bytes;                   // pointer to the first pixel to of the image
  u_int x_next, y_next;            // offset of the next pixel to return
  u_char *pixel_next;              // pointer to the next pixel to return
  size_t skip_bytes;               // number of bytes to skip when advancing
                                   // the pointer pixel_next from pixel
                                   // (x_end, i) to (x_start, i+1)
  // return the next pixel
  bool getNextPixel(u_int& x_off, u_int& y_off, svm_ImagePIxel& pixel);
};

svm_ImageIterator::svm_ImageIterator(u_int w, u_int h, u_int s, u_char* b,
                                     u_int x_s, u_int y_s, u_int x_e, u_int y_e)
   : x_start(x_s), y_start(y_s), x_end(x_e), y_end(y_e), pixelsize(s), bytes(b),
     x_next(x_s), y_next(y_s)
{
  if (x_end >= w) x_end = w-1;     // cut x_end back if x_end is out of range
  if (y_end >= h) y_end = h-1;     // cut y_end back if y_end is out of range
  pixel_next = bytes + (x_start + width * y_start) * pixelsize;
  skip_bytes = (width - x_end + x_start) * pixelsize;
}
```

Figure 42: Some SVM helper classes to be used by the customization code.

```
#include "t2_index.h"

class svm_ImageIndex: public T2_Index {
  T2_Array<FILE*> fps;
  u_int curfp;
};

T2_UDFRet
svm_ImageIndexConstructor(const T2_System& system, T2_UsrArgReader& arg,
                          const T2_Array<int>& fd, const T2_Array<int>& dsfd,
                          const T2_Array<T2_DSFileID>& dsfid,
                          T2_Index*& idxp)
{  // ignore system, arg
    idxp = (T2_Index *) new svm_ImageIndex(fd);
    return T2_UDFRet_OK;
}

svm_ImageIndex::svm_ImageIndex(T2_Array<int> fds)
{
  // turn all the file descriptors into file streams and store them in fps[]
  for (u_int i=0; i<fds.getNumberElements(); i++)
     fps[i] = fdopen(fds[i], "r");
}
```

Figure 43: The customized index class and its constructor function for the SVM application. Only the data member of the class are shown in the figure.

```
T2_UDFRet
svm_ImageIndex::fetchInit(const T2_System& system, const T2_Box& qr)
{
  curfp = 0;               // reset counter to start from first index file
  fseek(fps[curfp], 0, SEEK_SET);       // rewind the file
  return T2_UDFRet_OK;
}


T2_UDFRet
svm_ImageIndex::fetch(const T2_System& system, const T2_Box& qr,
                      bool& eod, T2_Box& mbr, T2_ClusterAuxInfoWriter& info,
                      T2_VArray<T2_BlockRequest>& chk)
{
  mbr.setNumberDimensions(2);    // each input is 2-dim
  while (curfp < fps.getNumberElements()) {
    u_int x_pos, y_pos, width, height, pixelsize, offset, size, fid;
    while (!fps[curfp].eof()) {
      // read from fps[curfp] all the stored information about this cluster
      my_read(fps[curfp], x_pos, y_pos, width, height, pixelsize, offset, size, fid);

      // compute mbr of the cluster
      mbr.getLow()[0] = x_pos;
      mbr.getLow()[1] = y_pos;
      mbr.getHigh()[0] = x_pos + width - 1;
      mbr.getHigh()[1] = y_pos + height - 1;

      if (qr ^ mbr) {           // cluster mbf intersects with query region
         // store x_pos and y_pos in info, just for demonstration purpose
         info.allocBuffer(sizeof(x_pos) + sizeof(y_pos));   // allocate buffer
         info << x_pos << y_pos;

         // set one block request for this cluster
         chk << T2_BlockRequest(fid, offset, size);

         eod = false;            // set end-of-data to false
         return T2_UDFRet_OK;
      }
    }           // while my_read()

    if (++curfp < fps.getNumberElements())
       fseek(fps[curfp], 0, SEEK_SET);       // rewind the file

  }             // while (curfp < fps.getNumberElements())

  eod = true;   // no more data
  return T2_UDFRet_OK;
}
```

Figure 44: The fetchInit() and fetch() functions of the customized index for the SVM example.

```
#include "t2_prj.h"

class svm_ImageProjection: public T2_ProjectFuncObj {
 u_int x_orig, y_orig, x_size, y_size;
};

T2_UDFRet
svm_ImgPrjConstructor(const T2_System& system, T2_UsrArgReader& arg,
                      T2_ProjectFuncObj*& prjfunc)
{
  arg >> xo >> yo >> xs >> ys;
  prjfunc = (T2_ProjectFuncObj *) new svm_ImageProjection(xo, yo, xs, ys);
  return T2_UDFRet_OK;
}
```

Figure 45: The customized class and its constructor function for attribute space service in the SVM application. Only the data members of the class are shown in the figure.

output attribute space is the coordinate space of lower-resolution image. In order to correctly project a coordinate in the input attribute space into the coordinate in the output attribute space, the user-defined projection function requires the coordinates of the top-left pixel of the output image, defined in the input attribute space, and the zoom values in both dimensions. These values come from the user argument, contained in the ADR query (see Figure 41). Figure 45 shows the definition of `svm_ImageProjection`, derived from the ADR base class `T2_ProjectFuncObj`. As is seen in the figure, `svm_ImageProjection` contains four data members to store the coordinates of the top-left pixel of the output image (`x_pos` and `y_pos`) and the zoom values; (`x_size` and `y_size`). The constructor function reads the four values out of the user argument, and uses them to initialize the projection function, `svm_ImageProjection`.

Figure 46 shows the implementation of the two virtual methods that `svm_ImageProjection` inherits from the `T2_ProjectFuncObj` base class. The method `svm_ImageProjection::project` projects an input point in the input attribute space to an output point in the output attribute space. This is done through the statements that assign values to `output[0]` and `output[1]`, as is shown in the figure. The method `svm_ImageProjection::project` sets the flag `succeed` to `true` only if the input point falls inside the given query box `iqr` and the output point it projects to falls inside the tile region `tilereg`. In this case, `project` adds the output point to the list of regions, specified by the argument `output_rg`. That is it effectively creates an output region that consists of one single point. Otherwise, the method sets the flag `succeed` to `false`, to indicate that either the input point is clipped out or it does not project to a point inside the accumulator tile region. The method `svm_ImageProjection::projectBox` simply projects the two diagonal corners of the input box, `inbox`, and uses the two resulting points as the corners of the output box, `outbox`.

B.2.5 Customization of Dataset Service

Figure 47 shows the customized classes for the dataset service. In particular, the SVM derived class `svm_ImageDataset` encapsulates a representation for an image by storing the width and height of the image, the number of bytes per pixel, and the maximum value stored with each byte. The SVM class `svm_ImageDatasetIterator` is an iterator for a data cluster of the dataset, and uses

```
T2_UDFRet
svm_ImageProjection::project(const T2_System& system,
                             const T2_Box& iqr, const T2_Region& tilereg,
                             const T2_Point& input_pt,
                             bool& succeed, T2_Region& output_rg)
{
  succeed = false;
  if (!iqr.contains(input_pt))          // point not inside query window
     return T2_UDFRet_OK;

  T2_Point output_pt(2);                // a point by projecting input_pt
  output_pt[0] = (input_pt[0] - x_orig) / x_size;
  output_pt[1] = (input_pt[1] - y_orig) / y_size;
  if (tilereg.contains(output_pt) == true) {
     output_rg.setNumberDimensions(2);
     output_rg.growMaxSetSize(1);       // grow region set to store one point
     output_rg << output_pt;
     succeed = true;
  }
  return T2_UDFRet_OK;
};

T2_UDFRet
svm_ImageProjection::projectBox(const T2_System& system, const T2_Box& inbox,
                                bool& succeed, T2_Box& outbox)
{
  outbox.setNumberDimensions(2);
  outbox.getLow()[0] = (inbox.getLow()[0] - x_orig) / x_size;
  outbox.getLow()[1] = (inbox.getLow()[1] - y_orig) / y_size;
  outbox.getHigh()[0] = (inbox.getHigh()[0] - x_orig) / x_size;
  outbox.getHigh()[1] = (inbox.getHigh()[1] - y_orig) / y_size;
  succeed = true;
  return T2_UDFRet_OK;
}
```

Figure 46: The project() and projectBox() functions of the customized attribute space service for the SVM application.

```
#include "t2_dataset.h"

class svm_ImageDataset: public T2_Dataset {
  u_int width, height, pixelsize, maxval;
}

class svm_ImageDatasetIterator: public svm_ImageIterator, public T2_Iterator {
  u_int x_pos, y_pos;
  svm_ImagePixel pixel;
};

T2_UDFRet
svm_ImageDatasetConstructor(const T2_System& system,
                            const T2_Array<int>& dfd, const T2_Array<int>& mdfd,
                            T2_Dataset*& dsp)
{
  // assuming that the aux data file has the information needed
  my_read(mdfd[0], width, height, pixelsize, maxval);
  dsp = (T2_Dataset *) new svm_ImageDataset(width, height, pixelsize, maxval);
  return T2_UDFRet_OK;
}
```

Figure 47: The customized dataset class, and its constructor function, and the dataset iterator class for the SVM application. Only the data member of the classes are shown in the figure.

the implementation from class `svm_ImageIterator` described in Section B.2.2. The additional information that class `svm_ImageDatasetIterator` keeps are the coordinates of the top-left corner of the entire image. The data member `pixel` of class `svm_ImageDatasetIterator` is used to represent the pixel the iterator returns.

Figure 48 shows the implementation of the method `svm_ImageDataset::genIterator`. This method computes the intersection between the minimum bounding rectangle of the cluster, `mbr`, and the input query box, `iqr`. The coordinates of the resulting rectangle is calculated relative to the first pixel of the cluster, and a `svm_ImageDatasetIterator` object is created.

Figure 49 shows the implementation for the virtual method `getNextElement` of the SVM class `svm_ImageDatasetIterator`. The method calls the `getNextElement` of the helper class `svm_ImageIterator`. The `eod`, which is the end of data flag, is set to `true`, if the method `svm_ImageIterator::getNextElement` returns false, to indicate that the last element has been accessed. Otherwise, the coordinates of the current pixel are computed, and the `eod` is set to `false`, to indicate that there may be more input elements (i.e. pixels) to be accessed in the data cluster.

B.2.6 Customizing Data Aggregation Service

Figure 50 shows the definitions of the customized accumulator class, the customized accumulator iterator class, and the customized accumulator meta-data object class. The accumulator meta-data object class `svm_ImageAccMeta` stores the width, height and the number of bytes per pixel for the entire output image, while the accumulator class `svm_ImageAcc` stores the image that corresponds to an output tile. The accumulator iterator class is implemented by the helper class

```
T2_UDFRet
svm_ImageDataset::genIterator(const T2_System& system, T2_IteratorID i,
                              const T2_Cluster& c,
                              const T2_Box& mbr, T2_ClusterAuxInfoReader& meta,
                              const T2_Box& iqr, const T2_Region& tilereg,
                              T2_ProjectFuncObj& prj, T2_Iterator*& iterp)
{
  // get the coordinates of the cluster's mbr and the input query box
  u_int mbr_x_low = (u_int) mbr.getLow()[0],
        mbr_y_low = (u_int) mbr.getLow()[1],
        mbr_x_high = (u_int) mbr.getHigh()[0],
        mbr_y_high = (u_int) mbr.getHigh()[1];
  u_int iqr_x_low = (u_int) iqr.getLow()[0],
        iqr_y_low = (u_int) iqr.getLow()[1],
        iqr_x_high = (u_int) iqr.getHigh()[0],
        iqr_y_high = (u_int) iqr.getHigh()[1];
  // find the set of pixels, relative to the first pixel of the cluster,
  // that the iterator needs to iterate over
  u_int x_start = (iqr_x_low > mbr_x_low)? (iqr_x_low - mbr_x_low) : 0,
        y_start = (iqr_y_low > mbr_y_low)? (iqr_y_low - mbr_y_low) : 0,
        x_end = (iqr_x_high > mbr_x_high)?
                        (mbr_x_high - mbr_x_low) : (iqr_x_high - mbr_x_low),
        y_end = (iqr_y_high > mbr_y_high)?
                        (mbr_y_high - mbr_y_low) : (iqr_y_high - mbr_y_low);
  iterp = (T2_Iterator *)
           new svm_ImageDatasetIterator(mbr_x_low, mbr_y_low,
                       mbr_x_high - mbr_x_low + 1, mbr_y_high - mbr_y_low + 1,
                       pixelsize, (u_char *) c.getDataPointer(0),
                       x_start, y_start, x_end, y_end);
  return T2_UDFRet_OK;
}
```

Figure 48: The customized genIterator() for the SVM application.

```
T2_UDFRet
svm_ImageDatasetIterator::getNextElement(const T2_System& system,
                const T2_Cluster& c, bool& eod, const void*& e, T2_Point& coord)
{
  if (svm_ImageIterator::getNextElement(x_off,  y_off, pixel) == false) {
     eod = true;                        // end of data
     return T2_UDFRet_OK;
  }

  coord.setNumberDimensions(2);       // compute coordinate of the pixel to return
  coord[0] = x_pos + x_off;
  coord[1] = y_pos + y_off;

  e = (const void *) &pixel;
  eod = false;
  return T2_UDFRet_OK;
}
```

Figure 49: The customized getNextElement() for the SVM application.

`svm_ImageIterator`, described in Section B.2.2.

The main purpose of the accumulator meta-data object is to partition the entire output image into tiles, and allocate each output tile when needed. Figure 51 shows the implementation of its two virtual methods, `stripMin` and `allocAcc`. The method `stripMine` partitions the entire output image by rows such that each tile contains a slice of the output image that fits in the memory space, the size of which is given in the `mem` parameter. The minimum bounding rectangle for each output tile is computed and added to the output argument `tile_mbrs`. The method `allocAcc` creates an image with the given minimum bounding rectangle `tile_mbr`.

Figure 52 shows the implementation of the two virtual methods of the `svm_ImageAcc` class. The method `navigateAll` is called when ADR initializes all the pixels in the accumulator, and therefore `navigateAll` simply returns an accumulator iterator that would iterate over all the pixels in the accumulator. The method `navigate` is called for each pixel in a data cluster retrieved during query processing. The argument `r` is a region obtained by projecting (via the `project` method implemented in the attribute space service) the coordinates of the pixel in the data cluster (returned by `svm_ImageDatasetIterator::getNextElement`) to the output attribute space. The implementation of method `navigate` computes the intersection between `r` and the minimum bounding rectangle of the accumulator, and returns a `svm_ImageAccIterator` that iterates through all pixels inside the resulting rectangle. Figure 53 shows the implementation of the method `svm_ImageAccIterator::getNextElement`, which returns the pixel inside a given rectangular region of the accumulator. Note that the implementation of this method is very similar to that of `svm_ImageDatasetIterator::getNextElement`, since both an input cluster and an accumulator region are basically 2-dimensional images.

Figure 54 shows the definition of the customized aggregation function, `svm_aggregation`. The two data members `packno` and `packtype` stores the packet type and the packet number to be used by the SVM client, as is described in Section B.2.1. The other data members are used during the global combine phase, and will be explained later. Figure 55 shows the two virtual methods that are used during the local reduction phase. The method `aifElem` takes a pixel in the accumulator

```
#include "t2_acc.h"

class svm_ImageAcc: public svm_Image, public T2_Accumulator {
  u_int x_res, y_res;        // width and height of entire output image
  u_int x_pos, y_pos;        // coordinates of the top-left pixel in the tile
};

class svm_ImageAccIterator: public svm_ImageIterator, public T2_AccIterator {
  svm_ImagePixel pixel;
};

class svm_ImageAccMeta: public T2_AccMetaObj {
  u_int width, height,        // size of the entire output image (in # pixels)
        pixelsize;            // number of bytes per pixel
};

T2_UDFRet
svmImageAccMetaConstructor(const T2_QueryInfo& qinfo, const T2_Box& output_box,
                           T2_UsrArgReader& arg, T2_AccMetaObj*& acmp)
{
  u_int w, h, psize;
  arg >> w >> h >> psize;
  acmp = (T2_AccMetaObj *) new svm_ImageAccMeta(w, h, psize);
  return T2_UDFRet_OK;
}
```

Figure 50: The customized accumulator class, the accumulator iterator class, and the accumulator meta-object class for the SVM application. Only the data members of the classes are shown.

```
T2_UDFRet
svm_ImageAccMeta::stripMine(const T2_QueryInfo& qinfo, size_t mem,
             const T2_ClusterInfoList& infolist, T2_VArray<T2_Region>& tile_mbrs)
{
  mem -= sizeof(svm_ImageAcc);      // reserve space for an svm_ImageAcc obj

  // for simplicity, we are just showing the code that partition the image by rows
  size_t rowsize = width * pixelsize;
  u_int y_mem = mem / rowsize,          // # rows that fit inside memory
        ntiles = height / y_mem;

  for (u_int i=0, y_start=0; i<ntiles; i++) {
     T2_Box mbr(2);
     T2_Point& low = mbr.getLow();
     T2_Point& high = mbr.getHigh();
     mbr.getLow()[0] = 0;
     mbr.getLow()[1] = (T2_ShapeCoord_t) y_start;
     mbr.getHigh()[0] = (T2_ShapeCoord_t) (width - 1);

     u_int y_end = y_start + y_mem - 1;
     if (y_end >= height)
        mbr.getHigh()[1] = (T2_ShapeCoord_t) (height - 1);
     else
        mbr.getHigh()[1] = (T2_ShapeCoord_t) y_end;

     tile_mbrs << mbr;
     y_start = y_end + 1;
  }
  return T2_UDFRet_OK;
}

T2_UDFRet
svm_ImageAccMeta::allocAcc(const T2_QueryInfo& qinfo, T2_Iteration i,
                           const T2_Region& tile_mbr, T2_Accumulator*& accp)
{
  const T2_Box& mbr = tile_mbr[0];

  u_int tile_width = (u_int) mbr.getHigh()[0] - (u_int) mbr.getLow()[0] + 1,
        tile_height = (u_int) mbr.getHigh()[1] - (u_int) mbr.getLow()[1] + 1;

  // always use rgb (3 bytes) per pixel, and max value per color component is 255
  const u_int pixelsize = 3,
              maxval = 255;
  accp = (T2_Accumulator *)
             new svm_ImageAcc(width, height,
                              (u_int) mbr.getLow()[0], (u_int) mbr.getLow()[1],
                               tile_width, tile_height, pixelsize, maxval);
  return T2_UDFRet_OK;
}
```

Figure 51: The implementation of allocAcc() and stripMine() for the customized accumulator meta-object class.

```
T2_UDFRet
svm_ImageAcc::navigateAll(const T2_QueryInfo& qinfo, T2_AccIterator*& aitp)
{
  aitp = (T2_AccIterator *)
        new svm_ImageAccIterator(getWidth(), getHeight(),
                                 getPixelSize(), getBytes(),
                                 0, 0, getWidth()-1, getHeight()-1);
  return T2_UDFRet_OK;
}

T2_UDFRet
svm_ImageAcc::navigate(const T2_QueryInfo& qinfo, T2_IteratorID i,
                       const T2_Region& r, T2_AccIterator*& aitp)
{
  const T2_Box& box = r[0];
  u_int x_low = (u_int) box.getLow()[0],
        y_low = (u_int) box.getLow()[1],
        x_high = (u_int) box.getHigh()[0],
        y_high = (u_int) box.getHigh()[1];

  t2_Assert (x_high >= x_pos);
  t2_Assert (y_high >= y_pos);

  // find the overlapping rectangle between box and the minimum bounding
  // rectangle of the accumulator
  u_int x_start = (x_low > x_pos)? (x_low - x_pos) : 0,
        y_start = (y_low > y_pos)? (y_low - y_pos) : 0,
        x_end = x_high - x_pos,
        y_end = y_high - y_pos;

  aitp = (T2_AccIterator *)
        new svm_ImageAccIterator(getWidth(), getHeight(),
                                 getPixelSize(), getBytes(),
                                 x_start, y_start, x_end, y_end);
  return T2_UDFRet_OK;
}
```

Figure 52: The implementation of navigateAll() and navigate() for the customized accumulator class.

```
T2_UDFRet
svm_ImageAccIterator::getNextElement(const T2_QueryInfo& qinfo,
                                     T2_Accumulator& acc,
                                     bool& eod, void*& elem)
{
  u_int x_off, y_off;                   // offsets for pixel to return

  if (svm_ImageIterator::getNextElement(x_off, y_off, pixel) == false) {
     // end-of-data is true
     eod = true;
     return T2_UDFRet_OK;
  }

  elem = (void *) &pixel;
  eod = false;
  return T2_UDFRet_OK;
}
```

Figure 53: The implementation for the getNextElement() for the customized accumulator iterator class.

tile and initializes all its bytes to zero. The `dafElem` method effectively performs a `max` operation. It takes an input pixel and an accumulator pixel, and sets the value of the accumulator pixel to the larger of the two values.

Figure 56 and Figure 57 show the implementation for the three virtual methods of the SVM class `svm_aggregation` for the global combine phase. Note that each ADR back-end node has a copy of the replicated accumulator, which is a part of the entire output image. The overall strategy in the global combine phase is to further partition the accumulator into sub-images by rows among the ADR back-end nodes, and have each ADR back-end node send the corresponding sub-images of the local copy of the accumulator to the corresponding ADR back-end nodes. Upon receiving a message, each ADR back-end node updates the pixels in the local subregion of the accumulator with the larger of the received and the local values.

The method `needGlobalCombine` returns `true` when there is more than one ADR back-end node running. In the case where there is only one ADR back-end node, the method simply assigns the entire accumulator to itself and returns `false` to end the global combine phase. If there are more than one ADR back-end node running, `fillAccMsgBuffer` partitions the accumulator into sub-regions (or sub-images), assign a sub-image to an ADR back-end node, and generates the messages that send the sub-images from the local copy of the accumulator to the corresponding ADR back-end nodes. The data members of class `svm_aggregation` are used to describe the sub-image assigned to the local processor. The data member `my_bytes` points into the accumulator pixel array assigned to the local node, `my_nbytes` specifies the size of the array in bytes. The data member `my_x_pos` and `my_y_pos` stores the coordinates of the top-left corner of the local sub-image, and data member `my_width` and `my_height` keep the width and the height. The `processAccMsg` method is called when a message arrives to aggregate (i.e., perform a max operation) the received pixel values with the pixel values stored in the local sub-image.

Figure 58 shows the implementation of the method `finalize`, which is responsible for generating an output object that would create the final image from the accumulator array in the local

```
#include "t2_aggr.h"

class svm_aggregation: public T2_AggregateFuncObj {
  u_int packno, packtype;       // packet number and type used by the SVM client
  u_char *my_bytes;             // the assigned bytes to combine on local proc
  size_t my_nbytes;             // the number of bytes assigned to local proc
  u_int my_x_pos, my_y_pos,     // the position of the 1st assigned pixel
                                // in the entire output image
        my_width,               // width of local assigned sub-image
        my_height;              // height of local assigned sub-image

};

T2_UDFRet
svm_aggregationConstructor(const T2_QueryInfo& qinfo, T2_UsrArgReader& arg,
                           T2_AggregateFuncObj*& afp)
{
  u_int packet_no, packet_type;
  arg >> packet_no >> packet_type;
  afp = (T2_AggregateFuncObj *) new svm_aggregation(packet_no, packet_type);
  return T2_UDFRet_OK;
}
```

Figure 54: The customized aggregation function object and its constructor for the SVM application. Only the data members of the class are shown.

```
T2_UDFRet
svm_aggregation::aifElem(const T2_QueryInfo& qinfo, void* accElem)
{
  t2_Assert (accElem != NULL);
  svm_ImagePixel& pixel = *((svm_ImagePixel *) accElem);
  for (u_int i=0; i<pixel.getNumberBytes(); i++)
     pixel[i] = 0;

  return T2_UDFRet_OK;
}

T2_UDFRet
svm_aggregation::dafElem(const T2_QueryInfo& qinfo, T2_DSID i, const void* dataElem,
                         void* accElem)
{
  const svm_ImagePixel& input_pixel = *((const svm_ImagePixel *) dataElem);
  svm_ImagePixel& output_pixel = *((svm_ImagePixel *) accElem);
  for (u_int i=0; i<input_pixel.getNumberBytes(); i++)
      if (input_pixel[i] > output_pixel[i])
         output_pixel[i] = input_pixel[i];
  return T2_UDFRet_OK;
}
```

Figure 55: The implementation for the functions of svm_aggregation to be used during the local reduction phase.

```
bool
svm_aggregation::needGlobalCombine(const T2_QueryInfo& qinfo, T2_Accumulator& acc)
{
  if (qinfo.getNumberNodes() > 1)
     return true;

  // only one processor, so no need to do global combine,
  // however, need to initialize the variables that describe the sub-image
  // since they were supposed to be set during fillAccMsgBuffer() if a global
  // combine phase was needed
  svm_ImageAcc *image = (svm_ImageAcc *) & acc;
  my_bytes = image->getBytes();
  my_nbytes = image->getNumberPixels() * image->getPixelSize();
  my_x_pos = image->getXPosition();
  my_y_pos = image->getYPosition();
  my_width = image->getWidth();
  my_height = image->getHeight();

  return false;
}

T2_UDFRet
svm_aggregation::processAccMsg(const T2_QueryInfo& qinfo, T2_ProcID sender,
                               T2_AccMsgBufferReader& msgbuf, T2_Accumulator& acc)
{
  for (u_int i=0; i<my_nbytes; i++) {
     u_char v;
     msgbuf >> v;
     if (my_bytes[i] < v)
        my_bytes[i] = v;
  }

  return T2_UDFRet_OK;
}
```

Figure 56: The implementation for the functions of svm_aggregation to be used during the global combine phase.

```
T2_UDFRet
svm_aggregation::fillAccMsgBuffer(const T2_QueryInfo& qinfo,
                                  T2_Accumulator& acc,
                                  T2_Array<T2_AccMsgBufferWriter>& msgbuf)
{
  u_int numnodes = qinfo.getNumberNodes();
  T2_ProcID mynode = qinfo.getMyNode();
  t2_Assert (numnodes > 1);
  t2_Assert (numnodes == msgbuf.getNumberElements());

  svm_ImageAcc *image = (svm_ImageAcc *) & acc;
  u_int width = image->getWidth(),
        height = image->getHeight(),
        pixelsize  = image->getPixelSize();
  u_char* bytes = image->getBytes();

  u_int y_part = height / numnodes,             // # rows per partition
        y_excess = height % numnodes;           // # excess rows
  u_int npixels_part = y_part * width;          // # pixels per partition
  size_t nbytes_part = npixels_part * pixelsize;   // # bytes per partition
  u_char* cur_bytes = bytes;
  for (u_int p=0, y=0; p<numnodes; p++) {
     u_int cur_y_part = y_part,
     cur_nbytes_part = nbytes_part;
     if (p == numnodes-1) {             // last processor gets all y_access rows
        cur_y_part += y_excess;
        cur_nbytes_part += y_excess * image->getPixelSize();
     }

     if (p == (u_int) mynode) {         // no msg to myself, but remember
                                        // my assigned bytes
        my_bytes = cur_bytes;
        my_nbytes = cur_nbytes_part;
        my_x_pos = image->getXPosition();
        my_y_pos = y + image->getYPosition();
        my_width = width;
        my_height = cur_y_part;
        cur_bytes += cur_nbytes_part;
    }
    else {                              // send a msg to remote proc p
        msgbuf[p].setUserDataBuffer((const char*) cur_bytes, cur_nbytes_part);
        cur_bytes += cur_nbytes_part;
    }                           // if (p == mynode) else

    y += cur_y_part;
  }                             // for (u_int p=0, y=0; p<numnodes; p++)

  return T2_UDFRet_OK;
}
```

Figure 57: The customized fillAccMsgBuffer() of svm_aggregation for the SVM application.

```
T2_UDFRet
svm_aggregation::finalize(const T2_QueryInfo& qinfo, T2_Accumulator& acc,
                          T2_Output*& outp)
{
  svm_ImageAcc *acc_image = (svm_ImageAcc *) &acc;

  // create the output sub-image object
  outp = (T2_Output *) new
        svm_OutputSubImage(packet_no, packet_type,
                           qinfo.getNumberNodes(), qinfo.getNumberTiles(),
                           my_x_pos, my_y_pos, my_width, my_height, my_bytes,
                           acc_image->getPixelSize(), acc_image->getMaxValue(),
                           acc_image->getTotalOutputImageWidth(),
                           acc_image->getTotalOutputImageHeight());
  return T2_UDFRet_OK;
}
```

Figure 58: The implementation for the functions of svm_aggregation to be used during the output-handling phase.

processor, and put the final image into buffers to be sent to the client. Figure 58 shows the definition of the `svm_OuputSubImage` class, which is derived from the ADR base class `T2_Output`. The `svm_OuputSubImage` class contains the following information:

- `packet_no` and `packet_type`: a packet number and a packet type, which will be used by the SVM client,
- `nprocs`: the number of ADR back-end nodes,
- `ntiles`: the number of output tiles,
- `x_res`, `y_res`: the width and height of the entire output image,
- `x_pos`, `y_pos`: the position of the sub-image in the entire output image,
- `nprocs`: the number of ADR back-end nodes,
- `sub_x_res`, `sub_y_res`: the width and height of the sub-image,
- `pixelsize`: the number of bytes per pixel,
- `maxval`: the maximum value of a byte for a pixel,
- `bytes`: the pixels of the sub-image.

These values are sent from the ADR back-end nodes to the SVM client, as shown in Figure 59. The number of processors and the number tiles tell the SVM client how many sub-images in total it should expect from the ADR back-end nodes. The position, the width and height of the sub-image allow the SVM client to stitch the various sub-images into one output image.

```
#include "t2_output.h"

class svm_OutputSubImage: public T2_Output {
  u_int packet_no,                      // packet number from aggregation func
        packet_type;                    // packet type from aggregation func

  u_int nprocs,                         // # back-end nodes
        ntiles,                         // # output tiles
        x_pos, y_pos,                   // position of the sub-image in the
                                        // entire output image
        sub_x_res, sub_y_res;           // width and height of sub-image
  u_char* bytes;                        // pointer to bytes of the sub-image
                                        // pixels (the bytes are owned by
                                        // another object, so no need to
                                        // delete them here)
  u_int pixelsize,                      // number of bytes per pixel
        maxval;                         // max value for each component byte
  u_int x_res, y_res;                   // width and height of entire output
};
```

Figure 59: The customized output class and its constructor for the SVM application. Only the data members of the class are shown.

B.3 Running the SVM Application

The ADR installation process (see Section 6.2) compiles the SVM customization code, and generates the following executables in the directory `example/svm/bin/`.

- `pbe`, the parallel ADR back-end program with the SVM customization code,
- `sbe`, the sequential ADR back-end program with the SVM customization code, and
- `svm-front-end`, the SVM application front-end.

The installation process also generates the following files.

- `catalogs/dataset-catalog`, the dataset catalog for the ADR back-end,
- `catalogs/index-catalog`, the index catalog for the ADR back-end,
- `front-end/constructor-catalog`, the constructor function catalog for the ADR front-end,
- `front-end/dataset-info`, the dataset information file for the ADR front-end,
- `chunkify-ppm`, a SVM utility program to partition a given PPM file into data chunks and load those data chunks into a ADR dataset,
- `gen-svm-qbatch`, a SVM utility program to create sample query files,
- `qbatch/qbatch*`, a set of sample SVM queries, generated by the SVM utility program `gen-svm-qbatch`.

```
size_t
svm_OutputSubImage::getOutputBufferSize(const T2_QueryInfo& qinfo)
{
  // need to allocate buffer for the header
  return sizeof(packet_no) + sizeof(packet_type)
         + sizeof(nprocs) + sizeof(ntiles)
         + sizeof(x_res) + sizeof(y_res)
         + sizeof(x_pos) + sizeof(y_pos)
         + sizeof(sub_x_res) + sizeof(sub_y_res);
}

T2_UDFRet
svm_OutputSubImage::flushOutput(const T2_QueryInfo& qinfo, T2_OutputBuffer& buf,
                                T2_OutputDataPtrList& ptr)
{
  const char *p1 = buf.getCurrentPosition(),    // pointer to header
             *p2;                               // a temp pointer

  // generate header of the output sub-image
  buf << nprocs << ntiles << x_res << y_res << x_pos << y_pos
      << sub_x_res << sub_y_res;

  p2 = buf.getCurrentPosition();
  ptr.insertDataPointer(p1, p2 - p1);           // insert pointer to header

  u_int npixels = sub_x_res*sub_y_res;
  if (npixels > 0)                              // insert pointer to the pixels
     ptr.insertDataPointer((const char*) bytes, npixels * pixelsize);

  return T2_UDFRet_OK;
}
```

Figure 60: The implementations of methods in the customized output class for the SVM application.

The SVM application is provided with three ADR datasets, stored in the directory `bin/datasets/`. Each dataset file of a dataset simulates a disk of a ADR back-end node. At run-time, each dataset file is assigned to a back-end node, and for a given query, all back-end nodes cooperate to read the appropriate chunks from the dataset files and process those chunks to generate the desired output image. We assume that the disk that contains the directory `bin/datasets/` is cross-mounted on all the machines that the parallel ADR back-end nodes run on. If this is not the case, then data files in `bin/datasets/*/` must be copied to the appropriate disks and the dataset registration file must be updated accordingly to specify the new paths to the dataset files and index files.

The SVM application can run in two ways.

- Run a sample query with the parallel ADR back-end alone.
  In this mode, the ADR back-end nodes would read a query from a given file, and each node would write its output into a file. A set of sample queries are created in the directory `bin/qbatch/` during the setup process. Each file in `bin/qbatch/` contains one single query about a particular dataset stored in `bin/datasets/`. At the end, each back-end node would generate a ppm file, and the users can view the files using their favorite image browser.

  To run the parallel ADR back-end with a sample query, do the following.

```
% pbe -r -d catalogs/dataset-catalog -i catalogs/index-catalog \
        -q <sample query file> -o ooo
```

  Here is a quick explanation of the various options.

  -r assigns the logical processor id's assumed in the back-end configuration to the actual ADR back-end nodes in a round- robin fashion; that is, assuming that there are in total <numnodes> back-end nodes, then logical processor id k is assigned to the back-end node with rank k % <numnodes>; this means that all data files assigned to logical processor id k in the dataset registration file are assigned to the back-end node with rank k

  -d specifies the dataset catalog file

  -i specifies the index catalog file

  -q specifies the sample query file (replace <sample query file> with a file in qbatch/)

  -o forces the back-end nodes to write output into files with the given prefix; the actual output file would be named as

```
<prefix>-<query number>-<tile number>.<processor rank>
```

  where <query number> is a running count of the number of queries that the back-end has processed so far, and <tile number> specifies the tile that the output corresponds to

  Note that some other parameters may be needed for MPI to start the parallel back-end appropriately, for example the number of MPI processes to run and the set of machines to run those processes on. If MPI-ch is used for the inter-processor communication layer, then the script program `bin/run-pbe-sample-query` can be used to start the parallel back-end. This script uses `mpirun` from MPI-ch to start up the parallel back-end, running over a given set of machines. Note that how the MPI processes are mapped to the give machines is entirely decided by MPI-ch.

```
% run-pbe-sample-query <sample query file> <number procs> \
                        [<host name 1> <host name 2> ...]
```

One could also run the sequential version in a similar way.

```
% run-sbe-sample-query <sample query file> <number procs>
```

- Run the full SVM application, which consists of a parallel back-end, a ADR front-end, an SVM application front-end, and an SVM client[1].

  This is a more realistic scenario for a customized ADR application. In this scenario, a ADR front-end process starts up first and awaits the connection of the parallel ADR back-end nodes. After all the back-end nodes have connected to the ADR front-end, the ADR front-end waits for the connection from an SVM application front-end. After connecting to the ADR front-end, the SVM application front-end inquires the ADR front-end for the datasets and functions registered for the svm application. The SVM application front-end then waits for the connection from an SVM client. An SVM client is the interface that users use to select regions of interest in an image and the desired resolution, as well as the browser that displays output images from queries. A selected region of an image and a resolution is transmitted from the client to the SVM application front-end, which converts the request into a ADR query and sends it to the ADR front-end. The ADR front-end forwards the query to the back-end nodes, which process the query and send the results back to the client for display.

  To run the full SVM application therefore requires one to start up four programs in the following order.

  1. Start the ADR front-end.

     ```
     % run-t2fe <number app front-ends> <back-end port> \
                <app front-end port>
     ```

     where <back-end port> is the port that the parallel ADR back-end nodes connect to, and <app front-end port> is the port that the SVM application front-end connects to. Though in practice the ADR front-end is intended to run all the time, in this simple example, the ADR front-end would terminate as soon as after <number app front-ends> SVM application front-ends have connected and disconnected from the ADR front-end.

     The ADR front-end uses the files in `bin/front-end/` to answer inquiries from the connected SVM application front-ends. It can interact with multiple SVM application front-ends at the same time.

  2. Start the parallel ADR back-end.

     ```
     % pbe -r -d catalogs/dataset-catalog -i catalogs/index-catalog \
              -f <ADR front-end host name> <back-end port>
     ```

[1] The example SVM client uses the Java Swing library and is implemented in Java 2. Therefore, Java 2 should already be installed to run the SVM client.

where -f specifies the host and the port number that the ADR front-end listens to for the back-end nodes. Again, more parameters may be needed to start up the parallel back-end program properly.

If MPI-ch is used, then the script `bin/run-pbe-with-t2fe` can be used to start the parallel ADR back-end node.

```
% run-pbe-with-t2fe <number procs> \
                    <ADR front-end host name> <back-end port> \
                    [<host name 1> <host name 2> ...]
```

The argument <ADR front-end host name> specifies the machine that the ADR front-end runs on, and <back-end port> is the port that the ADR front-end listens to. Note that <back-end port> should be the same <back-end port> used when the ADR front-end was started. If a list of host names are provided, then these names are passed to MPI-ch as the set of machines to run the parallel back-end processes on. MPI-ch decides how the back-end processes are assigned to the given set of machines.

Under MPI-ch an alternative way to specify the set of parallel ADR back-end processes is done via a "host file". The file `bin/sample-MPI-ch-hostfile` is a sample host file, which assigns four MPI processes to four machines, with the first one being processor 0, and specify for each MPI process the path to the executable. This gives the users more control over how MPI processes are assigned to processors. The script `bin/run-pbe-with-t2fe` can also pass such a host file to MPI-ch.

```
% run-pbe-with-t2fe <hostfile> \
                    <ADR front-end host name> <back-end port>
```

3. Start the SVM application front-end.

```
% svm-front-end -f <ADR front-end host name> <ADR front-end port> \
                -a <client port number>
```

or

```
% run-svm-front-end <ADR front-end host name> <ADR front-end port> \
                    <client port number>
```

where <ADR front-end host name> specifies the machine that the ADR front-end runs on, and <ADR front-end port> is the port that the ADR front-end listens to for the application front-end. Note that <ADR front-end port> should be the same <app front-end port> used when the ADR front-end was started. The argument <client port number> is the port number that an svm client should connect to. In the current implementation, each svm application front-end process can only serve one SVM client.

4. Start the SVM client.

```
% run-java-client <SVM front-end host name> <client port number>
```

The client is implemented in Java, and the compiled byte code is stored in directory `java-client/client.jar`. Therefore, the users must have `java` in their search path.

Also, the Java run-time system must support Swing. The argument <SVM front-end host name> specifies the machine that the SVM application front-end runs on, and <client port number> is the port that the SVM application front-end listens to. Note that <client port number> must be the same <client port number> used when the SVM front-end was started.

Once the four programs start and connect to each other, the users can select an image from the Java client, choose a region of interest in a thumbnail, select the desired resolution, and submit the query to ADR. The result of the query would then be displayed by the Java client in its display panel on the left.

B.4 Adding Datasets to the SVM Application

More datasets (ie images) can be added to the SVM application through the following steps. These images must be in the PPM format.

1. Partition the PPM image into multiple chunks, assign those chunks to multiple data files, and create an index for the dataset, using the SVM utility program `bin/chunkify-ppm`.

```
% chunkify-ppm <ppm file> \
               <number of x partitions> <number of y partitions> \
               <output type> \
               <number of dataset files> <dataset file prefix> \
               <index file prefix> <aux file prefix>
```

   This utility program partitions the PPM image into <number of x partitions> $\times$ <number of y partitions> rectangular chunks, creates <number of dataset files> data files, and assign the chunks to those files in a round-robin fashion. The data files will have filenames in the format of <dataset file prefix>.n, when n ranges from 0 to <number of dataset files>-1. These data files can be distributed to all the disks available in the system where the parallel T2 back-end runs. An auxiliary file, <dataset file prefix>.aux is also created to store the PPM header information, which includes the total image width, height, number of bytes per pixel, and the max byte value. Furthermore, One index file is created for each dataset file, and they have the filenames in the format of <index file prefix>.n, where n also ranges from 0 to <number of dataset files>-1. The index files should be assigned to the disks that their corresponding data files are assigned to. Store these files in a sub-directory under datasets/.

2. Create a thumbnail in the PPM format, using a program such as `xv`. Put the thumbnail into the same sub-directory that the data files live ine.

3. Pick an image name for the new dataset and add an entry to the dataset registration file, `bin/dataset-registration.txt`. The entry should contain the following fields:

```
t2-svm-example:<image nam>
t2-svm-example:image-dataset-constructor
<path to the thumbnail>
<image total width> <image total height>
d <path to data file 0>         0
d <path to data file 1>         1
```

```
            :
t2-svm-example:image-dataset-index
t2-svm-example:image-dataset-index-constructor
<some description or a blank line>
i <path to index file 0>        0 -1
i <path to index file 1>        1 -1

            :
===
```

4. Re-run the SVM setup process to update various catalogs.

```
% make setup
```

References

[1] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. Hollingsworth, J. Saltz, and A. Sussman. Tuning the performance of I/O-intensive parallel applications. In *Proceedings of the Fourth ACM Workshop on I/O in Parallel and Distributed Systems*, May 1996.

[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] 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.

[4] C. F. Cerco and T. Cole. User's guide to the CE-QUAL-ICM three-dimensional eutrophication model, release version 1.0. Technical Report EL-95-15, US Army Corps of Engineers Water Experiment Station, Vicksburg, MS, 1995.

[5] 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.

[6] 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.

[7] S. Chippada, C. N. Dawson, M. L. Martínez, and M. F. Wheeler. A Godunov-type finite volume method for the system of shallow water equations. *Computer Methods in Applied Mechanics and Engineering*, 1997. Also a TICAM Report 96-57, University of Texas, Austin.

[8] G. Edjlali, A. Sussman, and J. Saltz. Interoperability of data parallel runtime libraries. In *Proceedings of the 11th International Parallel Processing Symposium*. IEEE Computer Society Press, Apr. 1997.

[9] C. Faloutsos and P. Bhagwat. Declustering using fractals. In *Proccedings of the 2nd International Conference on Parallel and Distributed Information Systems*, pages 18–25, San Diego, CA, Jan. 1993.

[10] R. Ferreira, B. Moon, J. Humphries, A. Sussman, J. Saltz, R. Miller, and A. Demarzo. The Virtual Microscope. In *Proceedings of the 1997 AMIA Annual Fall Symposium*, pages 449–453. American Medical Informatics Association, Hanley and Belfus, Inc., Oct. 1997.

[11] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proceedings of the International Conference on Data Engineering*, pages 152–159, New Orleans, Louisiana, Feb. 1996.

[12] A. Guttman. R-Trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM-SIGMOD Conference*, pages 47–57, Boston, MA, June 1984.

[13] 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.

[14] R. A. Luettich, J. J. Westerink, and N. W. Scheffner. *ADCIRC*: An advanced three-dimensional circulation model for shelves, coasts, and estuaries. Technical Report 1, Department of the Army, U.S. Army Corps of Engineers, Washington, D.C. 20314-1000, December 1991.

[15] K.-L. Ma and Z. Zheng. 3D visualization of unsteady 2D airplane wake vortices. In *Proceedings of Visualization'94*, pages 124–31, Oct 1994.

[16] The Moderate Resolution Imaging Spectrometer. *http://ltpwww.gsfc.nasa.gov/MODIS/MODIS.html*.

[17] NASA Goddard Distributed Active Archive Center (DAAC). Advanced Very High Resolution Radiometer Global Area Coverage (AVHRR GAC) data. Available at http://daac.gsfc.nasa.gov/CAMPAIGN_DOCS/LAND_BIO/origins.html.

[18] G. Patnaik, K. Kailasnath, and E. Oran. Effect of gravity on flame instabilities in premixed gases. *AIAA Journal*, 29(12):2141–8, Dec 1991.

[19] C. T. Shock, C. Chang, B. Moon, A. Acharya, L. Davis, J. Saltz, and A. Sussman. The design and evaluation of a high-performance earth science database. *Parallel Computing*, 24(1):65–90, Jan. 1998.

[20] T. Tanaka. Configurations of the solar wind flow and magnetic field around the planets with no magnetic field: calculation by a new MHD. *Jounal of Geophysical Research*, 98(A10):17251–62, Oct 1993.

[21] The University of California, Berkeley. *The Gist C++ library, version 1.0*, 1996.

[22] U.S. Geological Survey. Land satellite (LANDSAT) thematic mapper (TM). Available at http://edcwww.cr.usgs.gov/nsdi/html/landsat_tm/landsat_tm.

[23] The USGS General Cartographic Transformation Package, version 2.0.2. *ftp://mapping.usgs.gov/pub/software/current_software/gctp/*, 1997.