An Approach for Automatic Data Virtualization *

Li Weng *, Gagan Agrawal*, Umit Catalyurek†, Tahsin Kurc †,
Sivaramakrishnan Narayanan†, Joel Saltz†
*Department of Computer and Information Sciences
†Department of Biomedical Informatics
Ohio State University
Columbus OH 43210

Abstract

Analysis of large and/or geographically distributed scientific datasets is emerging as a key component of grid computing. One challenge in this area is that scientific datasets are typically stored as binary or character flat-files, which makes specification of processing much harder. In view of this, there has been recent interest in data virtualization, *and* data services *to support such virtualization.*

This paper presents an approach for automatically creating data services to support data virtualization. Specifically, we show how a relational table like data abstraction can be supported for complex multi-dimensional scientific datasets that are resident on a cluster. We have designed and implemented a tool that processes SQL queries (with select and where statements) on multi-dimensional datasets. We have designed a meta-data description language that is used for specifying the data layout. From such description, our tool automatically generates efficient data subsetting and access functions.

We have extensively evaluated our system. The key observations from our experiments are as follows. First, our tool can correctly and efficiently handle a variety of different data layouts. Second, our system scales well as the number of nodes or the amount of data is scaled. Third, the performance of the automatically generated code for indexing and extracting functions is quite comparable to the performance of hand-written codes.

*This research was supported in part by the National Science Foundation under Grants #ACI-9619020 (UC Subcontract #10152408), #EIA-0121177, #ACI-0203846, #ACI-0234273, #ACI-0130437, #ANI-0330612, #ACI-9982087, Lawrence Livermore National Laboratory under Grant #B517095 (UC Subcontract #10184497), NIH NIBIB BISTI #P20EB000591, Ohio Board of Regents BRTTC #BRTT02-0003.

1 Introduction

Analysis of large and/or geographically distributed scientific datasets [5], is emerging as a key component of grid computing. A key challenge in this area is that scientific datasets are typically stored as binary or character flat-files. Such *low-level* layouts enable compact storage and efficient processing. Because the use of relational or other database technologies can result in significant storage overheads and slower processing, they have typically not been very popular in most scientific communities.

The use of low-level and specialized data formats, however, makes the specification of processing much harder. Recognizing this, several ongoing projects, such as BinX and Binary Format Description (BFD) [3], are proposing machine-interpretable descriptions of binary data layouts. Data Format Definition Language (DFDL) working group under the Global Grid Forum (GGF) is trying to standardize such efforts. While such proposals can allow precise description of the datasets in a remote repository, they do not alleviate the need for detailed understanding of the formats, or the dependence of an application on a particular low-level data layout.

In view of this, there has been recent interest in *data virtualization*, and *data services* to support such virtualization. In the mailing list of Global Grid Forum's DAIS working group, the following definitions were presented[1] *"A Data Virtualization describes an abstract view of data. A Data Service implements the mechanism to access and process data through the Data Virtualization".*

Using data virtualization and data services, low-level, compact, and/or specialized data formats can be hidden from the applications analyzing grid-based datasets. However, supporting data virtualization can require significant effort. For each dataset layout and abstract view that is desired, a set of data services need

[1] Please see http://www-unix.gridforum.org/mail_archive/dais-wg/Archive/msg00215.html

to be implemented. An additional challenge arises from the fact that the design and implementation of efficient data virtualization and data services oftentimes require interaction of two complementary players. The first player is the scientist who possesses a good understanding of the application, datasets, and their format, but is less knowledgeable about database and data services implementation. The second player is the database developer who is proficient in the tools and techniques for efficient database and data services implementation, but has little knowledge of the specific application.

This paper proposes a meta-data and compiler-oriented approach to facilitate a common meeting ground for the two players and to enable automatic creation of efficient data services to support data virtualization. Specifically, we show how a relational table like data abstraction can be supported for complex multi-dimensional scientific datasets that are resident on a cluster. By using a well-defined meta-data description language, the scientist and database developer together can describe the format of the datasets generated and used by the application. Using a compiler that can parse the meta-data description and generate code to navigate the datasets, the database developer (or the scientist) can conveniently generate data services that will serve the datasets.

There are two key aspects of our approach. First, we have designed a meta-data description language that can be used for describing a low-level data layout. This description language is expressive enough to allow: 1) dataset physical layout within the file system of a node, 2) dataset distribution on nodes of one or more clusters, 3) the relationship of the dataset to the logical or virtual schema that is desired, and 4) the index that can be used to make subsetting more efficient. Second, our tool automatically generates efficient data subsetting and access functions for a given meta-data description. These functions take the user query as input and help create relational tables. The runtime support for these functions is provided by a middleware, called STORM, that processes SQL queries (with SELECT and WHERE statements) on multi-dimensional datasets and provides services for data selection, data partitioning, and data transfer operations on a parallel system [10, 9].

Clearly, the virtualization we provide is also provided by relational and object-relational databases. For read-only and very large scientific datasets, our approach offers at least two significant advantages. First, the data can be kept in the original format it is generated from simulations or collected from instruments. Second, the time and storage overhead associated with loading the data in a database and managing it is not required.

We have extensively evaluated our system. The key observations from our experiments are as follows. First, our tool can correctly and efficiently handle a variety of different data layouts. Second, our system scales well as the number of nodes or the amount of data is scaled. Third, the performance of the automatically generated code for indexing and extracting functions is quite comparable to the performance of hand-written codes.

2 Overview of the System and Motivating Applications

In this section, we give an overview of our system. We also describe some of the applications that have motivated this work, and give details of the STORM runtime system we use.

2.1 System Overview

As we will further establish through some examples, scientific applications frequently involve large multi-dimensional datasets. Particularly, the data generated by scientific simulations or the data collected from scientific instruments involves spatial and temporal coordinates. Consider a scenario where such a dataset is hosted on a remote repository. Scientists across the grid will typically be interested in downloading a subset of a dataset. The criteria used for subsetting can include one or more of the following: 1) range of spatial and/or temporal coordinates, 2) parameters used for a specific simulation, 3) the set of attributes that are of interest, 4) value of one or more of the attributes of interest, and 5) the return value from a user-defined function applied to one or more of the attributes.

If a dataset is stored as a flat-file or a set of flat-files, a user will need to have a detailed understanding of the layout to be able to select the values of interest. The basic premise of our work is that a virtual relational table view and SQL queries on such a virtual view provide a very convenient yet powerful mechanism for specifying subsets of interest.

Figure 1 (left) shows the canonical structure of the queries. A specific example is shown in the right hand side. The *Data Elements* clause as part of the *Select* operation is used for specifying the attributes of interest. The *Expression* (after *WHERE*) can contain operations on ranges of values, whereas the *Filter(Data Element)* allows the use of application-specific and user-defined filter operations that are difficult to express with simple comparison operations. Since our goal is to support subsetting, we do not allow joins, aggregations, or group-by operations.

A high-level overview of our system is shown in Figure 2. The underlying runtime system we use, STORM, is described later in this section. The STORM system requires the users to provide an *index* function and an *extractor* function. The index function

SELECT < *Data Elements* >
FROM < *Dataset Name* >
WHERE < *Expression* > AND
Filter(< Data Element >)

SELECT *
FROM IparsData
WHERE RID in (0,6,26,27) AND TIME $\geq$ 1000 AND
TIME $\leq$ 1100 AND SOIL $\geq$ 0.7
AND SPEED(OILVX, OILVY, OILVZ) $\leq$ 30.0;

Figure 1. Canonical Query Structure (left) and An Example Query from IPARS Application (right)

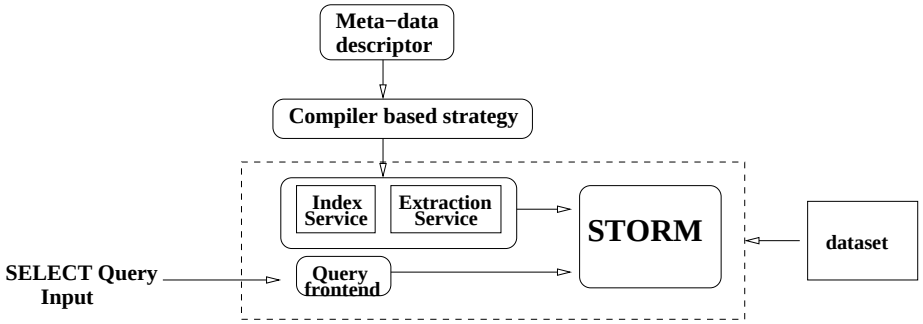


Figure 2. Overview of Our System

chooses the file chunks that need to be processed for a given query. The extractor is responsible for reading the file(s) and creating rows of the virtual table. Thus, the use of the STORM system itself requires significant knowledge of the file layout and the index that may be available.

2.2 Motivating Applications

This work is motivated by data-driven applications from science, engineering, and biomedicine. These applications include simulation-based studies for oil reservoir management, water contamination studies, cancer studies using Magnetic Resonance Imaging (MRI), telepathology with digitized slides, and analysis of satellite data. Here, we describe two applications that are used as case studies in this paper.

Oil Reservoir Management: Cost-effective and environmentally safer production of oil from reservoirs is only possible with effective oil reservoir management. A management strategy should integrate into the decision process a good understanding of physical properties of the reservoir. Although field instrumentation has been enhanced over the years, most of the time a partial knowledge of critical parameters such as rock permeability is available. Thus, complex numerical reservoir models are needed and it is essential that geological uncertainity be incorporated into these models. An approach is to simulate alternative production strategies (number, type, timing and location of wells) applied to realizations of multiple geostatistical models [12]. A typical study involves a large collection of simulations (also referred to as realizations) that model the effects of varying oil reservoir properties (e.g., permeability, oil/water ratio, etc.) over a long period of time. Simulations are carried out on a three-dimensional grid. At each time step, the value of seventeen separate variables and cell locations in 3-dimensional space are output for each cell in the grid. Each of the output variables are written to files. If the simulation is run in parallel, the data for different parts of the domain can reside on separate disks or nodes.

Large scale simulations can generate tens of Gigabytes of output per realization, resulting in Terabytes of data per study. Analysis of this data is key to achieve a better understanding and characterization of oil reservoirs. This can require access to subsets of data in a distributed environment. Common analysis scenarios involve queries for economic evaluation as well as technical evaluation, such as determination of representative realizations and identification of areas of bypassed oil. An example query is *"Find the largest bypassed oil regions between time T_1 and T_2 in realization A."*.

Satellite Data Processing: Analysis of data acquired by earth-orbiting satellites can provide valuable information about regional and global changes. A satellite dataset consists of a number of measurements by a satellite orbiting the earth continuously [4]. While the satellite passes over a region, its sensors record readings from the surface. Each measurement is a data element and is associated with a location (lat-

itude,longitude) on the surface and the time of recording. Five sensor values are stored with each data element. Therefore, a data element in a satellite dataset can be viewed as having 8 attributes (two spatial, one time dimension, and five sensors).

A query specifies a rectangular region and a time period. The query can also choose a subset of sensor readings. A typical analysis processes the data for up to a year and generates one or more composite images of the area under study. Generating a composite image requires projection of the globe onto a two dimensional grid; each pixel in the composite image is computed by selecting the "best" sensor value that maps to the associated grid point.

The raw data from satellites is processed to correct for drift in instrument calibration. The processed data is used to answer queries. Since common queries are range queries over space-time dimensions, the processed data is oftentimes partitioned into data chunks and stored as a set of chunks in order to improve query performance. Each chunk represents a subregion in the space-time domain and contains all the sensor readings whose coordinates fall into that subregion. A spatial index is built so that chunks that intersect the query are searched for quickly.

2.3 The STORM Runtime System

STORM is a middleware desgined to support data selection, data partitioning, and data transfer operations on flat-file datasets hosted on a parallel system [10, 9]. STORM is architected as a suite of loosely coupled services. The *query service* is the entry point for clients to submit queries to the database middleware. The *data source* service provides a view of a dataset to other services. It provides support for implementing application-specific *extraction* function. An extraction function returns an ordered list of attribute values for a tuple in the dataset, thus effectively creating a virtual table. The *indexing service* encapsulates indexes for a dataset, using an index function provided by the user. The *filtering service* is responsible for execution of user-defined filters. After the set of tuples that satisfy the query has been determined, the data should be partitioned among the processing units of the client program and transferred from the server to those processors. The purpose of the *partition generation service* is to make it possible for an application developer to implement the data distribution scheme employed in the client program at the server. The *data mover* service is responsible for transferring selected data elements to destination processors based on the partitioning description generated by the partition generation service.

The tool we describe in this paper is primarily responsible for automatically generating the index and extractor functions. To enable this, the layout of the data and the available index needs to be described in a systematic fashion by the administrator managing the data repository. For this purpose, we have designed a *meta-data description* language. This is presented in the next section. Section 4 describes how our tool automatically generates *index* and *extractor* functions.

3 Meta-data Description Language

This section describes the meta-data description language we use as part of our system.

3.1 Requirements and Overview

Our goal was to have a meta-data description language which is very expressive, and particularly, can allow description of: 1) dataset physical layout within the file system of a node, 2) dataset distribution on nodes of one or more clusters, 3) the relationship of the dataset to the logical or virtual schema that is desired, and 4) the index that can be used to make subsetting more efficient. In addition, we also wanted the language to be easy to use for data repository administrators, and to serve as a convenient basis for our code generation tool.

Though several notations already exist for meta-data description, none of them meet all of the above requirements. BinX and Binary Format Description (BFD) [3] are for describing a single file and do not conveniently allow description of index associated with a dataset. HDF5 [8] is a data storage format, and not a meta-data description. It requires data to be reformatted in a specific format, which can involve significant overheads for large scientific datasets. However, our language does use certain key-words and features from HDF5.

While our current implementation is specific to the meta-data description language we describe here, our basic approach can be used for supporting virtualization on top of datasets that use standards like HDF5, or individual files that use descriptions like BinX or BFD. Further, note that the description language we have developed can easily be embedded in an XML file and made machine independent. In our presentation, though, we focus on the concepts associated with the description language and do not use XML.

Our meta-data descriptor comprises three components.

1. Dataset Schema Description: states the logical or virtual relational table view that is desired.

2. Dataset Storage Description: lists the nodes and the directories on the system where the data is resident.

3. Dataset Layout Description: describes the actual layout of the data within and across different files.

The use of these three components in our system is shown in Figure 3.

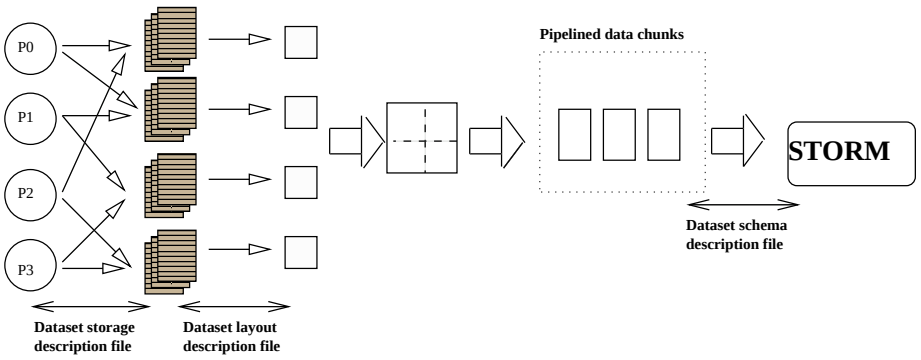


Figure 3. Use of the Three Components of Meta-data Descriptor in Our System

Component I: Dataset Schema Description

```
[IPARS]          // {* Dataset schema name *}
REL = short int   // {* Data type definition *}
TIME = int
X = float
Y = float
Z = float
SOIL = float
SGAS = float
```

Component II: Dataset Storage Description

```
[IparsData]      // {* Dataset name *}
// {* Dataset schema for IparsData *}
DatasetDescription = IPARS
DIR[0] = osu0/ipars
DIR[1] = osu1/ipars
DIR[2] = osu2/ipars
DIR[3] = osu3/ipars
```

Component III: Dataset Layout Description

```
DATASET "IparsData" { // {* Name for Dataset *}
  DATATYPE { IPARS } // {* Schema for Dataset *}
  DATAINDEX { REL TIME }
  DATA { DATASET ipars1 DATASET ipars2 }
  DATASET "ipars1" {
    DATASPACE {
      LOOP GRID ($DIRID*100+1):(($DIRID+1)*100):1 {
        X Y Z
      }
    }
    DATA { $DIR[$DIRID]/COORDS $DIRID = 0:3:1 }
  } // end of DATASET "ipars1"
  DATASET "ipars2" {
    DATASPACE {
      LOOP TIME 1:500:1 {
        LOOP GRID ($DIRID*100+1):(($DIRID+1)*100):1 {
          SOIL SGAS
        }
      }
    }
    DATA { $DIR[$DIRID]/DATA$REL $REL = 0:3:1 $DIRID = 0:3:1 }
  } // {* end of DATASET "ipars2" *}
}
```

Figure 4. The Meta-data Descriptor for the IPARS Dataset

3.2 Detailed Description and An Example

To further explain the three components of our description language, we use a running example based upon the IPARS dataset from oil reservoir simulation studies [12]. Here, the dataset comprises several simulations on the same grid, each involving a number of time-steps. These simulations are identified by a realization identifier (REL). The X, Y, and Z coordinates of each point in the grid is stored explicitly. For each realization, each time-step, and each grid point, a number of attributes or variables are stored in the dataset.

The physical layout we consider is as follows. We have a 4 node cluster. The grid is divided into four partitions, and each node stores values of all attributes for all time-steps and all realizations for one partition. The X, Y, and Z coordinates for the grid points are stored only once and in a separate file, called COORDS, as they do not change over time and realizations. For storing the values of attributes, a separate file is used for each realization. In each such file, the data is ordered by time-steps. For each time-step, we store the value of the two attributes (SOIL and SGAS)

for all grid points in the partition. The spatial coordinates of grid points are not stored explicitly in each file, instead, the values of attributes SOIL and SGAS are stored in the same order in which coordinates are stored in the file COORDS.

The meta-data description is shown in Figure 4. The first two components, the dataset schema and dataset storage, are quite simple. We focus our discussion on the dataset layout. This description is based upon the use of six key-words: DATASET, DATATYPE, DATAINDEX, DATASPACE, DATA, and LOOP. A DATASET is a nested structure, which can comprise of one or more other DATASETs. A DATASET can be described by using DATATYPE, DATAINDEX, DATASPACE, and DATA. DATATYPE can be used for relating a DATASET to a schema (as shown in Figure 4), or for defining new attributes that are not part of the schema. DATAINDEX is used for stating the attributes that can be used for indexing the data. DATASPACE is used for the leaf nodes in the structure, i.e., for DATASETs that do not comprise other DATASETs. It describes the layout associated with each file in the DATASET. For non-leaf nodes in the description, DATA is used for listing the DATASETs that are nested. For leaf nodes, DATA is used for listing the files.

In Figure 4, "IparsData" comprises "ipars1" and "ipars2". "ipars1" comprises a single file on each node, which stores the X, Y, and Z coordinates for the grid-points in the partition. Within a DATASPACE, the key-word LOOP is used for capturing the repetitive structure within a file. The variable $DIRID is used for identifying the directory. Thus, the clause "LOOP GRID ($DIRID*100+1):(($DIRID+1)*100):1" implies that we store X, Y, and Z coordinates for grid-points 1 through 100 in the file residing on directory 0 (DIR[0]), grid-points 101 through 200 in the file residing on directory 1 (DIR[1]), and so on. (The number of grid points on each node is identical in this example). The DATA field as part of "ipars1" shows that four different files are associated with this dataset, corresponding to the four different directories listed earlier.

Now, let us consider the "ipars2" dataset. Each file associated with this dataset stores the attributes SOIL and SGAS for 500 time-steps and 100 grid-points. The use of the same loop identifier GRID implies that values for these 100 grid-points are stored in the same order as in the file COORDS. This dataset comprises 16 files, corresponding to the four directories and four different RELs.

4 Automatic Virtualization Using Meta-Data

We now describe the key aspects of how we generate index and extraction class codes to be used by the STORM system. Using the meta-data and the query, the key data-structure we try to compute at runtime

```
Data_Extract {
  Find_File_Groups()
  Process_File_Groups()
}

Find_File_Groups {
  Let S be the set of files that match against the query
  Classify files in S by the set of attributes they have
  Let S_1, ..., S_m be the m sets
  T = φ
  foreach {s_1, ..., s_m} s_i ∈ S_i {
    {* cartesian product between S_1, ..., S_m *}
    If the values of implicit attributes are not inconsistent {
      T = T ∪ {s_1, ..., s_m}
    }
  }
  Output T
}

Process_File_Groups {
  foreach {s_1, ..., s_m} ∈ T
    Find_Aligned_File_Chunks()
    Supply implicit attributes for each file chunk
    foreach Aligned File Chunk {
      Check against index
      Compute offset and length
      Output the aligned file chunk
    }
  }
```

Figure 5. Data Extraction Analysis

is the set of *aligned file chunks* (AFC), each of which comprises

$$\{num\_rows, \{File_1, Offset_1, Num\_Bytes_1\}, \ldots,$$

$$\{File_m, Offset_m, Num\_Bytes_m\}\}$$

Here, num_rows denotes the number of rows of the table that can be computed using these file chunks. m is the number of chunks involved. A given set of AFCs contain only one chunk from each file; note that there may be multiple sets of AFCs from the same set of files. Thus, m is also equal to the number of files that are required to generate the table rows. For each file chunk, we store the file name, the offset at which we will start reading, and the number of bytes to be read from the file to create one row. By reading the m files simultaneously, with Num_Bytes_i bytes from the file $File_i$, we create one row of the table. Starting from the offset $Offset_i$, $num\_rows \times Num\_Bytes_i$ bytes are read from the file $File_i$. Aligned file chunks effectively correspond to a vertical partitioning of the table among the files that constitute the dataset. That is, a data file can contain data elements associated with a subset of table attributes. By determining the set of files and the set of file segments that are aligned with

each other, portions of the table can be constructed using the compiler generated data extraction functions.

The algorithm we use is presented in Figure 5. In our implementation, this algorithm is executed in two phases. First, our tool parses the available meta-data and generates code for indexing and data extraction functions. At runtime, these functions take the query as input and compute and read the set of AFCs. The advantage of this design is that the expensive processing associated with the meta-data does not need to be carried out at runtime. At the same time, no code generation or expensive runtime processing is required when a new query is submitted. For simplicity, we present a single algorithm that combines the two phases. The details of code generation are not presented in the paper.

There are two main steps in our algorithm. In the first step, we find sets of files which may need to be read together to compute rows of the relational table. In the second step, we find the aligned file chunks that must be retrieved from these sets of files. To explain our algorithm, we consider the meta-data description in Figure 4. The query we consider involves selecting a subset with REL values of 0 and 1, and TIME ranging from 1 to 100.

Initially, all files in the dataset are matched against the range query. It is determined if a file has data corresponding to the given query. For the example we are considering, files DATA2 and DATA3 (in all directories) will be excluded. This is because the file names are related to the REL values in our meta-data description. S is the set of files remaining after this analysis.

Next, we divide the files on the basis of the attributes whose value they store. In our example, the files COORD in the four directories are put in the same group, and the files DATA0 and DATA1 are put in another group. In this example, we have two groups, S_1 and S_2.

An important concept in our algorithm is *implicit attributes* associated with files or file chunks. Consider the file DIR[0]/DATA0. From the description of the dataset, we can see that the value of REL is 0, and the grid coordinates range from 1 to 100. Because these attribute values or value ranges are not stored explicitly, but instead inferred from the directory or file name and the meta-data description, they are referred to as implicit attributes.

Our algorithm now tries to determine the sets of files which can jointly contribute towards rows of a table. We consider each possible $\{s_1, \ldots, s_m\}$ such that $s_i \in S_i$. Then, we check if the value of the implicit parameters are consistent or not. For example, the range of grid coordinates for DIR[0]/COORD and DIR[1]/DATA0 are non-overlapping. Therefore, we know that they cannot be used together for creating a row of the relational table. The groups $\{s_1, \ldots, s_m\}$ that are not eliminated are put in the set T. In our example, eight such groups are put in the set T, which are {DIR[k]/COORD,DIR[k]/DATA0} and {DIR[k]/COORD,DIR[k]/DATA1 }, with k ranging from 0 to 3.

The next step in our algorithm involves determining aligned file chunks from the groups of files. Given an m file group $\{s_1, \ldots, s_m\} \in T$, our goal is to find file sections from each of the m files, which meet two criteria. First, their layouts must be identical. Second, the value of any implicit attributes should also be identical. Consider the set {DIR[0]/COORD,DIR[0]/DATA0}. The inner loop (over GRID) in DATA0 has the same layout as the file COORD. Therefore, the section of DATA0 corresponding to a specific TIME value forms an aligned file chunks with the file COORD. For our example, a total of 500 such aligned file chunk sets can be formed from each set in T. By using the query range, we can see that only 100 of these should be processed. Finally, we determine the file offsets, number of bytes to be read, and the number of rows that can be computed.

5 Experimental Results

We have carried out a number of experiments for evaluating our automatic data virtualization tool. Specifically, we had the following goals in designing our experiments: 1) Comparing our tool with an existing relational database (PostgreSQL) for read-only queries on large scientific datasets, 2) Testing the ability of our code generation tool to handle a variety of layouts of the same data, and evaluating the impact of layout on performance, 3) Evaluating the scalability of our tool as the number of data sources and size of the dataset is scaled, and 4) Comparing the performance of automatically generated indexing and extractor functions with the hand-written versions, whose performance was reported in earlier publications on STORM [9, 10].

The datasets and queries we use correspond to two applications, oil reservoir management (Ipars) and satellite data processing (Titan). Brief description of these applications was presented in Section 2.2. Our experiments were carried out on a Linux cluster where each node has a PIII 933MHz CPU, 512 MB main memory, and three 100GB IDE disks. The nodes are inter-connected via a Switched Fast Ethernet.

As listed above, our first goal was to compare our virtualization approach with the use of an existing relational database system. Note that the use of the former involves a significant overhead for loading the data and managing the database. However, our focus here is on evaluating the query processing time only. We used the Titan dataset with 6 GB raw data. The total storage required after loading the data in PostgreSQL was 18 GB. Figure 6 displays the execution

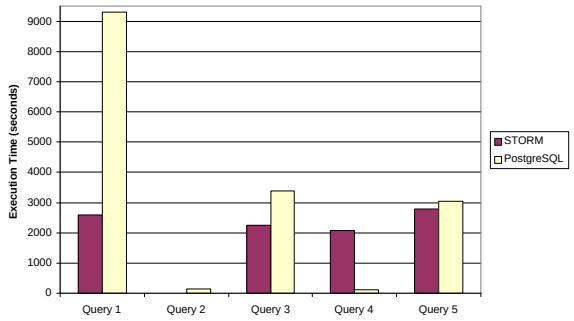


Figure 6. Comparison of PostgreSQL and STORM for Titan Dataset and Queries

time of the five queries listed in Figure 7. Data is indexed by spatial coordinates in both the systems and also by attribute S1 in PostgreSQL. Both the tools have been run using their default parameters, without any special tuning for this experiment.

Our tool has lower execution time for four of the five queries. Our tool especially performs much better on queries that requires the processing of large amount of data. For example, PostgreSQL takes about 9,300 seconds to execute Query 1, whereas the same query takes only 2,600 seconds using our system. PostgreSQL performs better than our system only when a small portion of the data is accessed directly via an index, which is the case for Query 4.

No.	Description
1	SELECT * FROM TITAN
2	SELECT * FROM TITAN WHERE X>=0 AND Y<=10000 AND Y>=0 AND Y<=10000 AND Z>=0 AND Z<=100
3	SELECT * FROM TITAN WHERE DISTANCE(X, Y, Z)<1000
4	SELECT * FROM TITAN WHERE S1 < 0.01
5	SELECT * FROM TITAN WHERE S1 < 0.5

Figure 7. Queries from Titan Dataset

Our previous work [9] also showed a similar performance gain for the queries on the Ipars dataset. The queries we used for that set of experiments are listed in Figure 8.

Our second set of experiments focused on evaluating our tool as the dataset layout is modified. Our goal was to demonstrate that our tool is able to correctly and efficiently handle a variety of different layouts for the same data. We used the Ipars dataset, the five queries listed in Figure 8, and the following six different file layouts:

- Layout I - All data is in one file, each tuple is stored as a record and each time step is organized as a chunk, i.e. the entire virtual table is stored in one file with the tuples sorted on time.
- Layout II - All data is in one file, which is organized with each time-step as a chunk. Within each chunk, each variable stored as an array.
- Layout III - Each time step is stored in a separate file and each file contains tuples in a tabular fashion.
- Layout IV - Each time step is stored in a separate file and each variable is stored as an array.
- Layout V - Data stored in 7 files where the first file has X, Y, and Z coordinate values, and the remaining attributes are divided between the remaining 6 files.
- Layout VI - Similar to Layout V, the attributes are partitioned in 7 files, but in each file, each variable is stored as an array.

Figure 9 shows the query execution times for the above file formats. We also used the original layout that was available to us from our application collaborators. In this layout, referred to as L0, all attributes are in different files. We compared the hand-written version for L0 format with the compiler generated versions for L0 and the six formats listed above. Since the execution time of the first query is an order of magnitude higher than the others, it has been displayed in a separate chart (Figure 9(a)).

Clearly, the execution time for all queries varies with the dataset layout. For the format L0, all queries involve opening 18 different files to compute one set of aligned file chunks, which can slow down the processing. Here, we can observe that the compiler generated code is only up to 10% slower than hand-written code. The difference is even less than 4% when a more complex select expression (involving a user-defined function) is used, e.g., in the Query 4. For all other layouts, the difference in execution times is very small, even though there are significant differences in how the data needs to be read. Overall, this experiment has shown that our tool can correctly and efficiently handle different layouts. The key advantage it provides is that index and extractor functions need not be hand-written for every new data layout.

Our final set of experiments focused on our two remaining objectives: comparing hand-written and automatically generated codes, and evaluating the system's scalability. The comparison of hand-written and compiler generated codes while scaling the number of nodes on which the data is hosted is shown in Figure 10. The query we have used for this experiment involves processing roughly 1.3 GB of Ipars data. The difference between compiler generated code and hand-written code varied between 5% to 34%, with an average difference of 16%. The execution times scaled almost linearly for both the versions.

The last experiment, Figure 11, compares the performance of compiler generated code against hand-written code while varying the query size. Figure 11(a) and Figure 11(b) displays the execution time of 4 different queries to Ipars and Titan datasets with increasing query sizes. Ipars data was partitioned on 16 nodes of the cluster, whereas the Titan data was stored on

Query No.	Type	Query Expression
1	Full scan of the table	SELECT * FROM IPARS
2	Subsetting using indexed attribute	SELECT * FROM IPARS WHERE TIME>1000 AND TIME<1100
3	Subsetting using indexed attribute and filtering	SELECT * FROM IPARS WHERE TIME>1000 AND TIME<1100 AND SOIL > 0.7
4	Subsetting using indexed attribute and filtering using a user defined function	SELECT * FROM IPARS WHERE TIME>1000 AND TIME<1100 AND Speed() < 30
5	Accessing the data from a remote client	SELECT * FROM IPARS WHERE TIME>1000 AND TIME<1050

Figure 8. Queries from Ipars Dataset

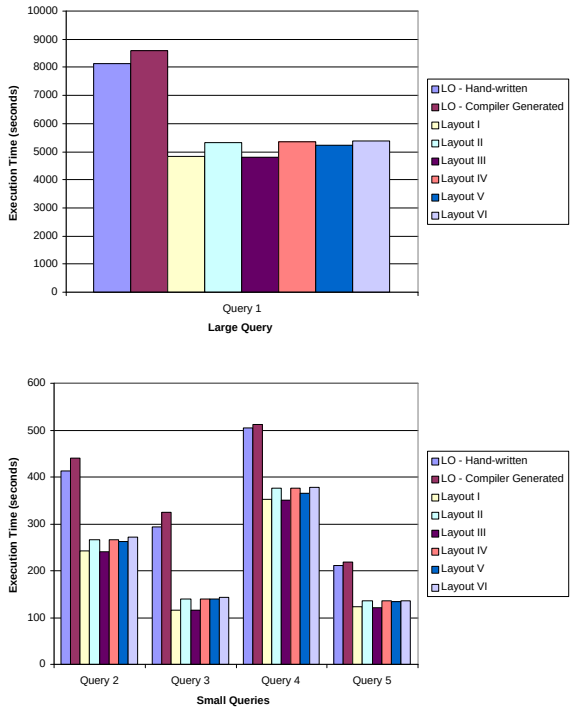


Figure 9. Query Execution Times Using Different File Layouts

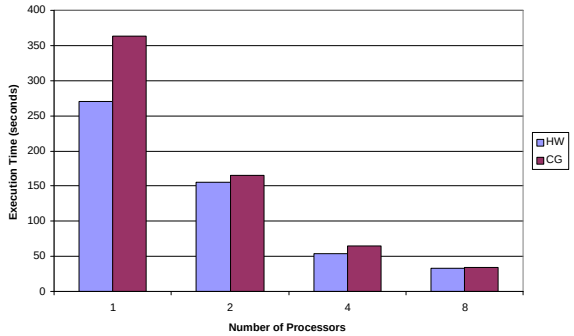
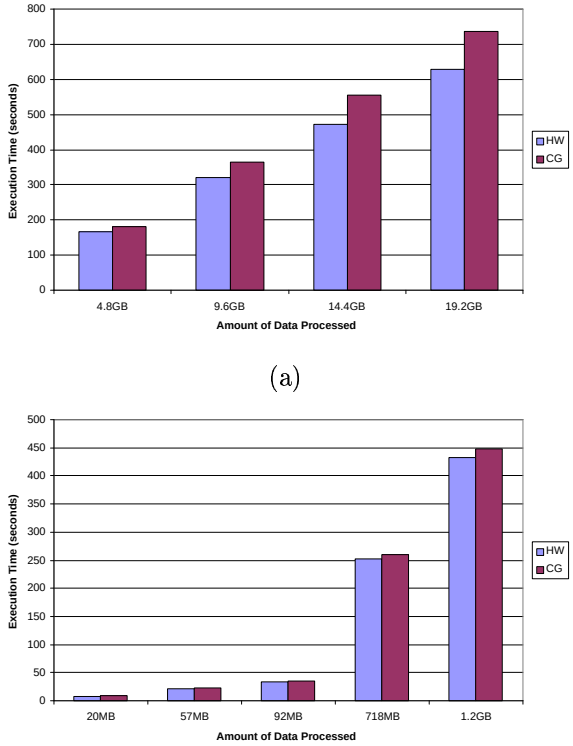


Figure 10. Scalability of the System with Increasing Data Sources



(a)

(b)

Figure 11. Execution Time With Varying Query Sizes: Ipars (a) and Titan (b)

a single node. For Ipars data, the performance difference between compiler generated code and hand written code was always within 17%, with an average difference of 14%. For the Titan data, the difference between compiler generated code and hand-written code is always within 4%. For both these applications, the processing time stays proportional to the amount of the data retrieved by the query.

6 Related Work

Our work has some relationships with previous efforts on meta-data descriptors, application-specific

data virtualization, and database systems.

BinX and Binary Format Description (BFD) [3] are meta-data descriptors to give a machine-readable view of a binary file. Our work is distinct in describing multiple files, including an index file, and the distribution of the datasets on multiple nodes. HDF5 [8] is a data storage format, and not a meta-data description.

There has been a lot of work on parallel, distributed, and grid-based databases, including support for multi-dimensional or spatio-temporal datasets. Sarawagi and Stonebraker showed how array chunks could be described and accessed as objects in an object-relational database [13]. The more recent work in database community treats multi-dimensional data as data cubes [14]. RasDaMan [2, 1] is a commercially available domain-independent DBMS for multi-dimensional arrays of arbitrary size and structure. Our work is distinct in supporting an abstract view of array-based datasets. Our work is also distinct from the work on parallel and distributed databases [11] in not requiring the datasets to be loaded in a specific system. Magda is a system built on top of MySQL for supporting Schemas on geographically distributed and replicated databases[2]. The support for *external tables* as part of Oracle's recent implementation allows tables stored in flat-files to be accessed from a database[3]. The data must be stored in the table format, or an *access driver* must be written. Also, there is no support for indexing such data.

OPeNDAP [6] provides data virtualization through a data access protocol and data representation. However, this system requires that the datasets be converted into a specific internal representation. SRS [7] is a system specific to bio-informatics that uses a special script language, called Icarus, to describe the contents of a file.

7 Conclusions

This paper has focused on the problem of supporting a rich class of data subsetting operations on remote data repositories that store scientific datasets in low-level formats. With the current state-of-the-art, there are two possible approaches to this problem. One involves manually implementing layout-specific data services and the other requires the dataset to be loaded into an existing and general-purpose database system.

We have presented an automatic data virtualization approach, in which a tool parses the meta-data information, and generates data extraction code. This code processes SQL queries with Select and Where expressions and creates virtual tables. We believe that this approach offers significant advantages. The data could be stored in the format it is generated in. The overheads and the effort involved in loading the data in a database system is avoided. At the same time, handling a new dataset layout or virtual view only involves writing a new meta-data descriptor. Experimental evaluation of our tool has demonstrated that this approach can handle a variety of different layouts, can scale to large datasets and large parallel systems, and provide performance that is competitive with hand-written codes.

[2]See www.atlasgrid.bnl.gov/magda/info

[3]See www.dbasupport.com/oracle/ora9i/External_Tables9i.shtml

References

[1] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system rasdaman. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 575–577. ACM Press, 1998.

[2] P. Baumann, P. Furtado, and R. Ritsch. Geo/environmental and medical data management in the RasDaMan system. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB97)*, pages 548–552, Aug. 1997.

[3] R. Baxter, R. Carroll, D. J. Ecklund, B. Gibbins, D. Virdee, and Q. Wen. BinX - A Tool for Retreiving, Searching, and Transforming Structured Binary Files. Available at http://www.edikit.org/binx/pub.htm, 2003.

[4] 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.

[5] A. Chervenak, I. Foster, C. Kesselman, C. Salisbusy, and S. Tuecke. The Data Grid: Towards An Architecture for the Distributed Management and Analysis of Large Scientific Datasets. *Journal of Network and Computer Applications*, 2001.

[6] P. Cornillon, J. Callagher, and T. Sgouros. OpeNDAP: Accessing data in a distributed, heterogeneous environment. *Data Science Journal*, 2:164–174, November 2003.

[7] T. Etzold and P. Argos. SRS: Information retrieval system for molecular biology data banks. *Methods in Enzymology*, 266:114–128, 1996.

[8] HDF Group at NCSA. Introduction to HDF5 Release 1.4. Available at http://hdf.ncsa.uiuc.edu/HDF5/doc/H5.intro.html, 2003.

[9] S. Narayanan, U. Catalyurek, T. Kurc, X. Zhang, and J. Saltz. Applying database support for large scale data driven science in distributed environments. In *Proceedings of the Fourth International Workshop on Grid Computing (Grid 2003)*, pages 141–148, Phoenix, Arizona, Nov 2003.

[10] S. Narayanan, T. Kurc, U. Catalyurek, and J. Saltz. Database support for data-driven scientific applications in the grid. *Parallel Processing Letters*, 13(2):245–271, 2003.

[11] M. T. Ozsu and P. Valduriez. Distributed and Parallel Database Systems. *ACM Computing Surveys*, 28(1):125–128, March 1996.

[12] J. Saltz, U. Catalyurek, T. Kurc, M. Gray, S. Hastings, S. Langella, S. Narayanan, R. Martino, S. Bryant, M. Peszynska, M. Wheeler, A. Sussman, M. Beynon, C. Hansen, D. Stredney, , and D. Sessanna. Driving scientific applications by data in distributed environments. In *Dynamic Data Driven Application Systems Workshop, held jointly with ICCS 2003*, Melbourne, Australia, June 2003.

[13] S. Sarawagi and M. Stonebraker. Efficient organizations of large multidimensional arrays. In *Proceedings of the Tenth International Conference on Data Engineering*, February 1994.

[14] C. Stolte, D. Tang, and P. Hanrahan. Polaris: a system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, Jan/Mar 2002.