Applying Database Support for Large Scale Data Driven Science in Distributed Environments*

Sivaramakrishnan Narayanan, Umit Catalyurek, Tahsin Kurc, Xi Zhang, Joel Saltz
Department of Biomedical Informatics, The Ohio State University
Columbus, OH, 43210
{krishnan,umit,kurc,xizhang,jsaltz}@bmi.osu.edu

Abstract

There is a rapidly growing set of applications, referred to as data driven *applications, in which analysis of large amounts of data drives the next steps taken by the scientist, e.g., running new simulations, doing additional measurements, extending the analysis to larger data collections. Critical steps in data analysis are to extract the data of interest from large and potentially distributed datasets and to move it from storage clusters to compute clusters for processing. We have developed a middleware framework, called GridDB-Lite, that is designed to efficiently support these two steps. In this paper, we describe the application of GridDB-Lite in large scale oil reservoir simulation studies and experimentally evaluate several optimizations that can be employed in the GridDB-Lite runtime system.*

1. Introduction

Cheaper disk space and faster networks allow large scale scientific applications to create dispersed repositories of large datasets. While computational requirements of many applications are still a challenge, applications that make use of large, distributed datasets have also emerged as major consumers of resources [3, 11]. In these applications, analysis of data *drives* the next steps taken by the scientist, e.g., running new simulations, doing additional measurements, extending the analysis to larger data collections. Optimizing reservoir management through simulation-based studies is an example of such *data driven* applications [15]. Despite technological advances in methods of determining reservoir properties, only a partial knowledge of critical parameters such as rock permeability, which govern production rates, are known from field measurements. Hence, a major challenge is to incorporate such geological uncertainty in large, detailed flow models. One approach to this problem is to simulate alternative production strategies (number, type, timing and location of wells) applied to multiple realizations of multiple geostatistical models. In a typical study, a scientist runs an ensemble of simulations to study the effects of varying oil reservoir properties (e.g., permeability, oil/water ratio, etc.) over a long period of time. Such an approach is highly data driven. Choosing the next set of simulations to be performed requires analysis of data from earlier simulations.

A critical step in analysis of data is to extract the data of interest from large and potentially distributed datasets and move it from storage clusters to compute clusters for processing. In [13], we presented the architectural design of a middleware framework, GridDB-Lite, that provides basic database support for data subsetting and data transfer operations in a Grid environment. This paper differs from our previous work in that we describe and evaluate the application of GridDB-Lite in the analysis of data from large scale oil reservoir simulation studies. We present a number of optimizations that can be applied in the GridDB-Lite infrastructure and experimentally evaluate them using large oil reservoir simulation datasets on a distributed collection of clusters.

2. Related Work

Parallel database systems have been a major research area in the database community [6, 12]. The database management community has also developed federated database technologies to provide unified access to diverse and distributed data. Most of the earlier efforts, however, have focused on relational models and closely integrated systems. Our framework can leverage techniques developed for those systems. It differs from previous work in that we target scientific datasets, where data is stored in files in application specific formats. In order to use a database management system, a scientific dataset should be loaded into the system since many database systems reorganize the data into their internal format. Our approach is to employ virtual tables through the use of data extraction objects. Moreover, the framework is designed as a set of loosely coupled services, which is more suitable for execution in distributed, heterogeneous environments. There are a few middleware

* This research was supported by the National Science Foundation under Grants #ACI-9619020 (UC Subcontract #10152408), #EIA-0121177, #ACI-0203846, #ACI-0130437, #ACI-9982087, Lawrence Livermore National Laboratory under Grant #B500288 and #B517095 (UC Subcontract #10184497).

```
SELECT $< Attributes >$
        FROM $Dataset_1, Dataset_2, ..., Dataset_n$
        WHERE $< Expression >$
            AND $Filter(< Attributes >)$
        GROUP-BY-PROCESSOR
            $ComputeAttribute(< Attributes >)$
```

Figure 1. Formulation of data retrieval steps as an object-relational database query.

systems designed for remote access to scientific data. Distributed Oceanographic Data System (DODS) [7] is one example. DODS is similar to our framework in that it allows access to user-defined subsets of data. However, the GridDB-Lite framework differs from DODS in that it builds on a more loosely coupled set of services that are designed to allow distributed execution of various phases of query evaluation (e.g., subsetting and user-defined filtering).

Several run-time support libraries and parallel file systems have been developed to support efficient I/O in a parallel environment [17, 4]. These systems mainly focus on supporting strided access to uniformly distributed datasets, such as images, maps, and dense multi-dimensional arrays. Our work, on the other hand, focuses on efficiently supporting subsetting and data movement operations over subsets of large datasets in a Grid environment.

There are some recent efforts to develop Grid services [9] and Web services implementations of database technologies [5]. Raman et. al. [14] discusses a number of *virtualization* services to make data management and access transparent to Grid applications. These services provide support for access to distributed datasets, dynamic discovery of data sources, and collaboration. Smith et. al. [16] address the distributed execution of queries in a Grid environment. They describe an object-oriented database prototype running on Globus and MPICH-G [10].

3. GridDB-Lite Middleware Framework

In this section, we briefly describe the Grid Database-Lite (GridDB-Lite) middleware [13] that is designed to support data select, data partitioning, and data transfer operations through an object-relational database model. In order to provide common support for a range of applications, a level of abstraction is necessary to separate application specific characteristics from the middleware framework. We develop a data subsetting model based on three abstractions: *virtual tables*, *select queries*, and *distributed arrays (or distributed data descriptors)*.

Virtual Tables. Datasets generated in scientific applications are usually stored as a set of flat files. However, a dataset can be viewed as a table. A tuple consists of a set of attribute values. The columns and rows of the table correspond to attributes and tuples, respectively.

Select Queries. With an object-relational view of scientific datasets, the data access structure of an application can be thought of as a *SELECT* operation as shown in Figure 1. The $< Expression >$ statement can contain operations on ranges of values and joins between two or more datasets. $Filter$ allows implementation of user-defined operations that are difficult to express with simple comparison operations.

Distributed Arrays. The client program that carries out the data processing can be a data parallel program implemented using programing paradigms such as MPI, High-performance Fortran, or KeLP [8]. The distribution of tuples in these paradigms can be represented as a *distributed array*, where each array entry stores a tuple. This abstraction is incorporated into our model by the *GROUP-BY-PROCESSOR* operation in the query formulation. $ComputeAttribute$ is another user-defined function that generates the attribute value on which the selected tuples are grouped together based on the application specific partitioning of tuples.

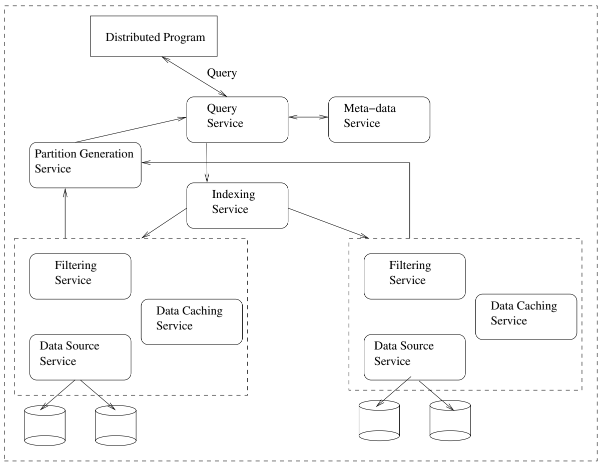
3.1. System Services

GridDB-Lite is organized as a set of coupled services as shown in Figure 2.

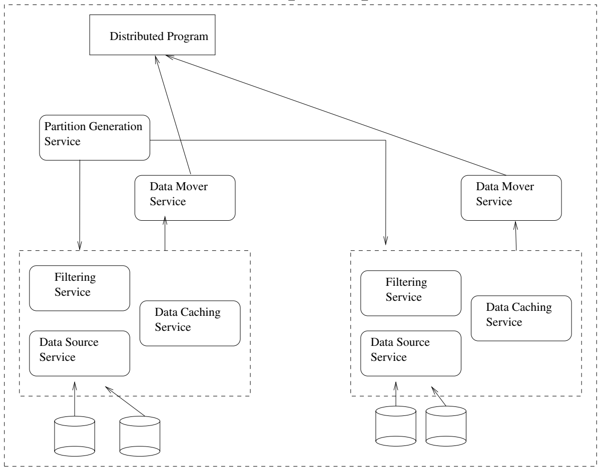
The **query service** provides an entry point for clients to submit queries to the database middleware. It is responsible for parsing the client query to determine which services should be instantiated to answer the query. The query service also coordinates the execution of services and the flow of data and control information among the services.

The **meta-data service** maintains information about datasets, and indexes and user-defined filters associated with the datasets. A scientific dataset is composed of a set of data files. The data files may be distributed across multiple storage units. The meta-data for a dataset can, for example, consist of a user-defined type for the dataset (e.g., MRI data, satellite data, oil reservoir simulation data), the name of the dataset, the list of attributes and attribute types (e.g., integer, float, composite) for a tuple in the dataset, and a list of tuples of the form *(filename,hostname)*, where *filename* is the name of the data file and *hostname* is the name of the machine on which the data file is stored.

Datasets in scientific applications are usually stored in flat files, and file formats vary widely across different application domains. This requires that either a dataset be converted into a common representation and stored in the database in that format or *virtual tables* be created when the dataset and its elements are accessed. With the latter approach, applications can access their data directly when desired. The **data source** service provides a view of a dataset in the form of virtual tables to other services. It provides support for implementing application specific *extraction*



(a) The inspector phase.



(b) The executor phase.

Figure 2. GridDB-Lite Architecture with a possible instantiation of its phases.

objects. An extraction object returns an ordered list of attribute values for a tuple in the dataset, thus effectively creating a virtual table.

Efficient execution of select operations is supported by two services; **indexing service** and **filtering service**. The **indexing service** encapsulates indexes for a dataset. With an index, tuples that satisfy the query can be searched quickly. A select query may contain user-defined filters and operations on attributes that are not indexed. To support such select operations, GridDB-Lite provides a **filtering service** that is responsible for execution of user-defined filters and the $< Expression >$ statements in Figure 1.

After the set of tuples that satisfy the query has been determined, the data should be partitioned among the processing units of the compute nodes where the data analysis program executes. The distribution of data among processors can be described by a distributed data descriptor as defined, for example, by KeLP [8] and HPF. Alternatively, a common data partitioning algorithm can be employed. The purpose of the **partition generation service** is to make it possible for an application developer to export the data distribution scheme employed in the data analysis program. In some cases, the distribution of data among the processors can be expressed in an application-specific compact form. In those cases, the partition generation service computes parameters for a partitioning function that is invoked during data transfer to client. The **data mover** service provides a *planner* function that computes an I/O and communication schedule to achieve high I/O bandwidth and minimize communication overheads. This schedule is passed to *data movers* components which are responsible for actually moving the tuples to destination processors.

3.2. Execution of a Query

In GridDB-Lite, query execution is carried out in two phases; an *inspector* phase and an *executor* phase. The goal of the inspector phase is to perform the steps needed to compute a schedule for moving the data from storage nodes to destination processors. The executor phase carries out the plan generated in the inspector phase to move tuples from source machines to the destination processors.

Step 1. Upon receiving a query, an index lookup is performed. Index returns information needed to extract tuples from each data source by extraction objects in the data source service. We classify attributes into three categories: 1) attributes that are used to determine whether the select predicate is satisfied (*select attributes*), 2) attributes that are used to determine how the query result will be partitioned among the destination processors (*partition attributes*), and 3) other attributes that are part of the result returned to the client program (*result attributes*).

Step 2. Extraction objects are executed on nodes where the dataset is stored. A table is generated that associates each extracted tuple with a unique ID, the values of select attributes, and the values of partition attributes.

Step 3. The table entries are streamed to the filtering service to determine which tuples satisfy the select predicate. The select filters executed by the filtering service remove the tuples that do not satisfy the predicate; they also strip the values of the select attributes. After the filtering operation, a table, referred to as *filtered planning table*, is created that contains only tuple unique IDs and partition attributes.

Step 4. The filtered planning table is streamed from the filtering service to the partition generation service to determine the partitioning of result tuples among the processors of the client program. The partition generation service associates each unique ID with a processor or computes parameters for a partition function to be invoked by the data mover service.

Step 5. Information from the partition generation service and the filtering service is sent to the data mover service.

The data mover service uses this information to compute an I/O and communication schedule and move data. The data mover service makes use of the extraction objects to extract the result attributes. The result attributes are then transfered to the client program.

Steps 1, 2, 3, and 4 correspond to the inspector phase, while step 5 is performed in the executor phase. Figure 2 shows a possible instantiation of the services and the execution of a query through the inspector and executor phases.

3.3. Optimizing Query Evaluation

The execution time of queries can be reduced by applying a number of optimizations. In this section, we look at a few optimization points in the overall system.

Colocating Services. The filtering service can be co-located with the data source service on the machines where the dataset is stored in order to reduce the volume of wide-area data transfer. If the partition generation service does not require data aggregation to determine partitioning of tuples, it can be co-located with the filtering and data source services on the same processor, cluster, or SMP.

Inspection using Data Chunks. The main drawback of using individual tuples in the inspection phase is the number of tuples can be very large, resulting in long execution times. In many scientific applications, datasets can be partitioned into a set of data chunks, each of which contains a subset of tuples. Each chunk also can be associated with metadata. An example of metadata would be a spatial bounding box. The inspection phase can be carried out using the data chunks. In this approach, the index is searched to find the data chunks that are selected by the query. The group-by-processor operation is executed on data chunks. When using data chunks, *upper bounds* on how many tuples will be sent to each processor and how much space will be needed at each processor are determined at the end of the inspection phase.

Distributed Execution of Filtering Operations. Both data and task parallelism can be employed to execute filtering operations in a distributed manner. If a select expression contains multiple user-defined filters, a network of filters can be formed. Individual filters can be placed on different platforms to minimize communication and computation overheads. In addition, multiple copies of an expensive filter can be created and executed to achieve data parallelism.

Data Caching. Caching the results from the filtering service can speed up the execution of data transfer, as cached results can be directly used to extract *select attributes* from data sources without going through the filtering service again. The filtered planning and group-by-processor tables can be cached when sufficient memory and/or disk space is available so that they can be re-used by the data mover service in step 5 of the query execution. Moreover, results generated by a query can be used by other queries in multi-query loads [1].

Parallel Data Transfer. Data is transferred from multiple data sources to multiple destination processors by the data mover service. Data movers can be instantiated on multiple storage units and destination processors to achieve parallelism during data transfer.

4. GridDB-Lite Implementation

We have developed a prototype implementation of GridDB-Lite using DataCutter [2], which is a component-based middleware framework designed to support processing of large multi-dimensional datasets in a distributed environment. We have chosen DataCutter as the underlying runtime system for the prototype for two reasons. First, it supports a filter-stream programming model for executing application-specific processing as a set of components, referred to as *filters*. Filters are connected via *logical streams*. A *stream* denotes a uni-directional data flow from one filter (i.e., the producer) to another (i.e., the consumer). The overall processing can be realized by a *filter group*, which is a set of filters connected through logical streams. Processing, network and data copying overheads are minimized by the ability to place filters on different platforms. This capability easily enables execution of various services and user-defined filters in a distributed environment. Second, in the DataCutter project, we are developing interfaces to the Globus, SRB, and NWS toolkits. This allows us to readily use the security, remote file access, resource monitoring and allocation services provided by these toolkits.

In order to develop a new database for an application, two base interface functions provided by GridDB-Lite have to be implemented by the database developer: *Index* and *Extractor*. Given a query, the responsibility of the *Index* is to return instances of *ChunkInfo* object which should contain necessary information to retrieve the tuples using instance of an *Extractor* object. The data source service is implemented using DataCutter filters. Instances of *Index* and *Extractor* objects are instantiated by *GridDBIndex* and *GridDBExtractor* filters. These filters extract the tuples and pass them to *GridDBFiltering* filter. The default evaluator object implemented in *GridDBFiltering* filter is capable of evaluating basic arithmetic operations as well as boolean operations. Additionally, database developers can register *user-defined functions* to be used for filtering of data. A database developer can also register a *DataPartitioner* function to perform application specific data partitioning. The output of *DataPartitioner* function is an instance of user-defined *Mapping* function. *GridDBDataMover* filters uses the instance of *Mapping* to compute the destination processors of the selected tuple. By default, GridDB-Lite provides a data partitioner and mapping function that perform round-robin assignment of tuples to processors.

Applications implemented using DataCutter consists of filters and a console process. The console process is used to interact with clients and coordinate the execution of application filters. In our prototype, when a query is received, first the indexing service is invoked. After index lookup, instance of *GridDBExtractor* are instantiated on the machines where the datasets are stored, and indexing results are streamed to those filters. *GridDBFiltering* filters are connected to the output streams of *GridDBExtractor*. The output of *GridDBFiltering* is connected to *GridDBDataPartitioner*. These filters form a filter group that is executed in the inspector phase. In this implementation, instead of implementing a mechanism that uses unique tuple IDs, we decided to apply filtering during the executor phase, for the sake of simplicity in the implementation. Therefore, after the inspector phase is completed, *GridDBExtractor*, *GridDBFiltering* and *GridDBDataMover* filters are instantiated. These filters form the filter group to move the tuples selected by a query to destination machines.

5. Supporting Analysis of Oil Reservoir Simulation Output

The main goal of oil reservoir management is to provide cost-effective and environmentally safer production of oil from reservoirs. A good understanding and monitoring of changing fluid and rock properties in the reservoir is necessary for the effective design and evaluation of management strategies. Since a partial knowledge of critical parameters such as rock permeability is available, it is desirable in production management to incorporate geologic uncertainty in complex reservoir models. An approach is to simulate alternative production strategies (number, type, timing and location of wells) applied to multiple realizations of multiple geostatistical models. This approach is highly data-driven. Choosing the next set of simulations to be performed requires analysis of data from earlier simulations [15].

In a typical study, a scientist runs a large collection of simulations, each of which is referred to as a *realization*, to study the effects of varying oil reservoir properties (e.g., permeability, oil/water ratio, etc.) over a long period of time. Large scale simulations can generate tens of Gigabytes of output per realization, resulting in Terabytes of data per study. Moreover, the datasets can be located at different sites, where the simulations are executed. This requires access to subsets of data in a distributed environment. Various data analysis operations can be carried out that query and manipulate the output datasets to forecast production amount, assess the economic value of the reservoir, and understand changing reservoir characteristics through seismic imaging and visualization. In this section, we describe the structure of oil reservoir simulation datasets and present the implementation of bypassed oil using GridDB-Lite.

5.1. Simulation Output

Simulations are carried out on a three-dimensional grid over several time steps. Each realization corresponds to different geostatistical models and different number of wells and well placements. The geostatistical models are used to generate permeability fields that are characterized by statistical parameters such as covariance and correlation length. 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 separate files. If the simulation is run in parallel, the data for different parts of the domain can reside on separate nodes.

5.2. Bypassed Oil Analysis

Depending on the distribution of reservoir permeability and the production strategy employed, it is possible for oil to remain unextracted from certain regions in the reservoir. To optimize the production strategy, it is useful to know the location and size of these regions of bypassed oil. To locate these regions, we apply the following algorithm [15]. In this algorithm, the user selects a subset of realizations, a subset of time steps ($[T_{start}, T_{end}]$), minimum oil saturation value ($SOIL_{tol}$), maximum oil speed ($Speed_{tol}$), and minimum number of connected grid cells (N) for a bypassed oil pocket. The goal is to find all the datasets that have bypassed oil pockets with at least N grid cells.

1. Find all oil cells in a dataset at time step $T \in [T_{start}, T_{end}]$, where $SOIL > SOIL_{tol}$ and $Speed_{oil} < Speed_{tol}$. Mark these cells as bypassed oil cells.
2. Run a connected components analysis on the selected cells at T to find pockets of cells that have more than N cells.
3. Perform an AND operation on all pockets of cells over all time steps in $[T_{start}, T_{end}]$. This results in pockets of cells that remain unchanged over time.
4. Run a connected components analysis on the result of the AND operation to find final pockets of cells that are bypassed oil regions.

This scenario accesses the large four-dimensional (three spatial dimensions and time) datasets which are output for each realization. Each of the output variables are written to separate files, so this computation involves the subsetting of data spread across several files.

5.3. Query Formulation

In this analysis scenario, the client request specifies a range of time steps and a set of realizations. Step 1 of the algorithm extracts the bypassed oil cells from the datasets and time steps requested by the client. Step 1 can be viewed as a select operation. The selected tuples are sent to the client program which executes steps 2-4 to find the bypassed oil regions. Figure 3 shows the formulation of step 1 of the algorithm as a query. In this figure, the result of the select operation is a list of tuples, each of which consists of the spatial coordinates of cells, realization id, and time step. These

```
SELECT $R.Cell_x, R.Cell_y, R.Cell_z, R.Id, R.Time$
    FROM $Realization_1, Realization_2, ..., Realization_n$
    WHERE $T_{start} <= R.Time$ AND $R.Time <= T_{end}$
        AND $R.SOIL > SOIL_{tol}$
        AND $Speed(R.V_{oil,x}, R.V_{oil,y}, R.V_{oil,z}) < Speed_{tol}$
    GROUP-BY-PROCESSOR $Partition(R.Id, R.Time)$
```

Figure 3. Formulation of data retrieval steps in the bypassed oil scenario as a query.

tuples are selected from the list of realizations specified by the client request (i.e., the FROM statement). A cell is selected if the expression in the WHERE statement is true. The selected tuples are partitioned among the client nodes using the user-defined Partition function in the GROUP-BY-PROCESSOR statement.

5.4. Implementation using GridDB-Lite

Virtual Tables. Each realization outputs seventeen variables at each grid node (cell) at each time step. These variables include oil, gas, and water saturation, the pressures of gas, oil, and water, and the velocities in each dimension of the grid of oil and water. In our implementation, the whole dataset has been treated as one big virtual table. The attributes of the virtual table are the seventeen variables, cell coordinates, time step, and realization id.

An extraction object was implemented in the data source service to retrieve the virtual table attributes. Given a list of data files comprising the output of the realization specified in the query, the extraction object returns the select attributes (oil saturation, oil velocity, time step), partition attributes (realization id and time step), and result attributes (cell coordinates, realization id, and time step).

Indexing Service. In data files, the values of the seventeen variables are grouped by time steps and stored in consecutive locations in the corresponding data file. An extraction object needs to know the number of variables stored per time step (i.e., the size of the grid used for that realization) and the starting offset of the time step in a data file. An index was implemented that, given a list of realizations and a range of time steps, returns the ids of the realizations that satisfy the query, a list of time steps per realization, and the size and offset of each time step in data files.

Filtering Service. For a bypassed oil cell, the speed of oil in that cell should be less than a user-specified threshold, $Speed_{tol}$. This is expressed as a user-defined filtering operation in Figure 3. Function $Speed(...)$ was implemented and registered with the filtering service. The function takes the velocity of oil in each dimension of the grid in a cell as input arguments and computes the speed of oil.

Partition Generation Service. A user defined Partition function was implemented in the partition generation service. This function returns a processor id, given a time step and realization id. We should note that computation of bypassed oil regions in steps 2-4 of the bypassed oil algorithm requires that all the cells at a time step in a realization be on the same processing node. Thus, the partition generation service computes a partitioning of the selected tuples based on the realization id and time step. The tuples with the same realization id and time step are grouped and assigned to a processing node on the client machine. The assignment of groups to processing nodes is done in a round robin fashion.

6. Experimental Results

In this section, we evaluate the performance of the GridDB-Lite implementation of bypassed oil analysis and examine the performance impact of several runtime optimizations. The hardware configuration used for the experiments consists of three Linux PC clusters. The first cluster, **OSU**, is made up of 24 Linux PCs. Each node has a PIII 933MHz CPU, 512MB main memory, and three 100GB IDE disks. The nodes are inter-connected via Switched Fast Ethernet. The second cluster, **OSC**, consists of 128 Linux PCs, each with dual 1.4GHz AMD Athlon CPUs, 2GB main memory and 70GB disks. The nodes are inter-connected via 2Gbps Myrinet 2000 switch. Both **OSU** and **OSC** clusters are hosted at the Ohio Supercomputer Center. The third cluster, **UMD**, is located at the University of Maryland. This cluster is composed of 50 Pentium III 650MHz processors. Each node has 768MB memory and two 75GB IDE disks and is connected to other nodes by Switched Fast Ethernet. The two sites, Maryland and Ohio, are connected to each other over a 100Mbps wide-area network. In the experiments, we executed queries accessing subsets of a dataset that consists of 1 Terabytes of data stored on the **OSU** and **UMD** clusters. This dataset was generated from 100 realizations executed on these two clusters. Each realization consists of simulation outputs of a grid with $65,536 = 64 \times 64 \times 16$ cells. At each time step, the value of seventeen separate variables is output for each node in the grid. A total of 2,000 time steps are taken and the total output stored for each realization is about 10 GB.

In the first set of experiments, we examine the effect on performance of distributed execution of filtering operations. The results are shown in Figure 4. The query in this experiment requested data over 10 time steps in 4 realizations stored across 16 nodes of the **OSU** cluster. We varied the number of filters from 1 to 22. As seen in the figure, increasing the number of filters decreases the execution time of the query. However, after an optimal point, creating more copies of filters does not improve the performance, and overhead of starting additional filters increases the overall execution time.

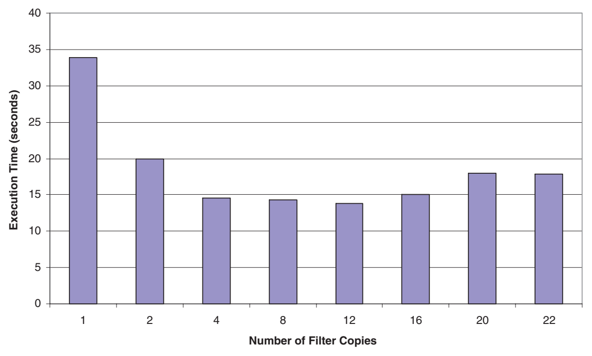


Figure 4. Effect of executing multiple filter copies.

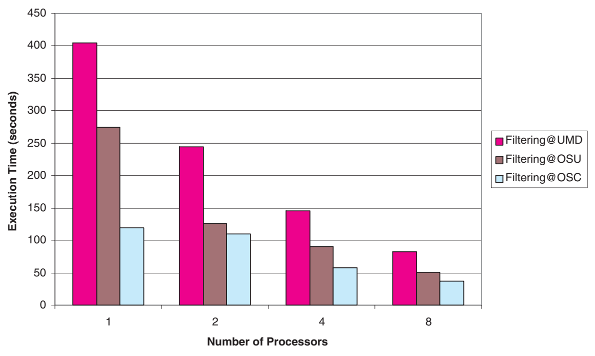


Figure 6. Effect of declustering of dataset and placement of filtering operations.

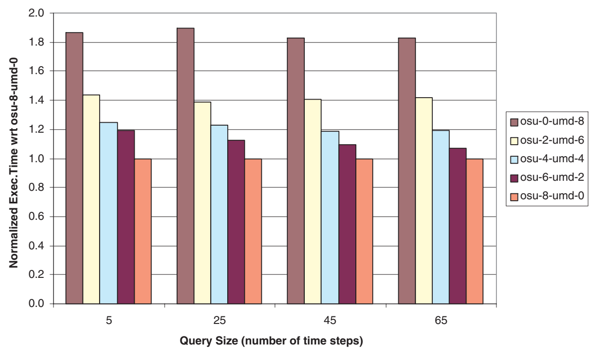


Figure 5. Colocating the Filtering Service with the Data Source Service.

The second set of experiments examines the effect of colocating the filtering service (i.e., user-defined filters in the query) with the data source service. In these experiments, the query requested four datasets stored on the 8 nodes of the **OSU** cluster. The number of time steps specified by the query was varied from 5 to 65. Figure 5 shows the query execution time, when the user-defined filters are placed on the **OSU** and **UMD** clusters. The total number of filter copies was fixed at 8 and each filter was executed on a separate processor. In the figure, *osu-x-umd-y* denotes that x filters are placed on the **OSU** cluster and y filters are placed on the **UMD** cluster. As expected, colocating the filters with the data source service reduces the query execution time significantly. In this experiment, the client program was executed on the **UMD** cluster. By running the user-defined filters near data sources, the volume of communication between the two clusters is reduced, resulting in lower query execution time.

Figure 6 shows the execution time of a query when a dataset is declustered across different number of storage nodes and when additional CPU resources can be used for filtering operations. In this set of experiments, the query requested data from a single dataset stored on the **OSU** cluster. The declustering of the dataset among the storage nodes was varied, and a single data extraction filter, *GridDBExtractor*, was executed on each storage node with data. The client processes were placed on the **UMD** cluster and the number of *GridDBFiltering* filters was set to match the number of the *GridDBExtractor* filters. We looked at three different placements of the *GridDBFiltering* and *GridDBDataMover* filters. In the first configuration, the *GridDBFiltering* and *GridDBDataMover* filters were placed on the **UMD** cluster. In the second configuration, they were placed on the **OSU** cluster. In the third configuration, the *GridDBFiltering* filters were placed on the **OSC** cluster, but the *GridDBDataMover* are placed on the **OSU** cluster, which has direct connection to outside networks via shared 100Mbps WAN. As is seen in the figure, the second configuration performs better than the first one. Clearly, colocating the filtering service with the data source service reduces the execution time. Moreover, GridDB-Lite can take advantage of faster CPUs and networks that are closely connected to data sources. Instantiating the filters on the **OSC** nodes further reduces the execution time, even though the filtered data has to be sent to the **OSU** nodes so that it can be transfered to the client processes running on the **UMD** cluster.

In the next set of experiments, we present a preliminary performance comparison of PostgreSQL and GridDB-Lite. In this experiment, we loaded one of the realizations, referred to here as RID0, to a PostgreSQL database. PostgreSQL (version 7.3.4) was run on one of the nodes of **OSU** cluster. We used default settings of both PostgreSQL and GridDB-Lite. The loading of the realization took more than 4 hours (14,711 seconds), and creating an index on TIME also took another 2 hours. The following queries were executed using both PostgreSQL and GridDB-Lite on one node of the cluster.

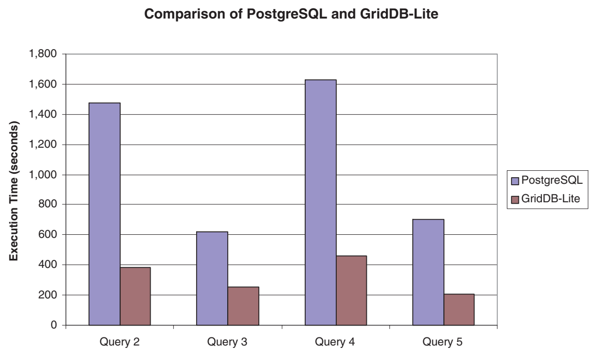


Figure 7. Comparison of PostgreSQL and GridDB-Lite.

- Query 1 - Full scan of the table: "SELECT * FROM RID0",
- Query 2 - Subsetting using indexed attribute: "SELECT * FROM RID0 WHERE TIME>1000 AND TIME<1100",
- Query 3 - Subsetting using indexed attribute and filtering: "SELECT * FROM RID0 WHERE TIME>1000 AND TIME<1100 AND SOIL > 0.7",
- Query 4 - Subsetting using indexed attribute and filtering using a user defined function: "SELECT * FROM RID0 WHERE TIME>1000 AND TIME<1100 AND Speed() < 30",
- Query 5 - Accessing the data from a remote client: "SELECT * FROM RID0 WHERE TIME>1000 AND TIME<1050",

Figure 7 displays the execution time of queries 2, 3, 4 and 5. Query 1 took 8,958 seconds using GridDB-Lite, and 18,685 seconds using PostgreSQL – since Query 1 took order of magnitude longer than the other queries, we do not include the execution time of the query in the figure. Our preliminary results show GridDB-Lite were able to execute the queries 23% to 74% faster than PostgreSQL. Since both systems provide multiple ways of optimizing query performance, we will further investigate the effect on performance of the optimizations for both systems.

We also compared the performance of the GridDB-Lite based implementation to the original implementation using DataCutter [15]. The experimental results show that the execution time of the GridDB-Lite version is only slightly (3.7%) slower than the hand optimized version of the bypassed oil analysis implementation using DataCutter.

7. Conclusions

We presented the application of a middleware infrastructure, GridDB-Lite, in analysis of datasets from large scale oil reservoir simulations. As shown by the experimental results, the loosely coupled services based design of GridDB-Lite makes it possible to easily incorporate a number of optimizations that reduce query execution time. We plan to extend this work to evaluate the application of GridDB-Lite for supporting applications on systems with multiple storage hierarchies.

References

[1] H. Andrade, T. Kurc, U. Catalyurek, A. Sussman, and J. Saltz. Persistent caching in a multiple query optimization framework. In *Proceedings of the Sixth Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*. Springer-Verlag, Mar. 2002.

[2] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, Oct. 2001.

[3] Biomedical Informatics Research Network (BIRN). http://www.nbirn.net.

[4] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Oct. 2000.

[5] Data Access and Integration Services. //http://www.cs.man.ac.uk/grid-db/documents.html.

[6] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[7] Distributed Oceanographic Data System. http://www.unidata.ucar.edu/packages/dods/.

[8] S. Fink, S. Kohn, and S. Baden. Efficient run-time support for irregular block-structured applications. *Journal of Parallel and Distributed Computing*, 50(1):61–82, Apr. 1998.

[9] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid services for distributed system integration. *IEEE Computer*, 36(6):37–46, June 2002. Open Grid Services Architecture (OGSA).

[10] The Globus Project. http://www.globus.org.

[11] Grid Physics Network (GriPhyN). http://www.griphyn.org.

[12] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, Dec. 2000.

[13] S. Narayanan, T. Kurc, U. Catalyurek, and J. Saltz. Database support for data-driven scientific applications in the grid. *Parallel Processing Letters*, 2003. To appear.

[14] V. Raman, I. Narang, C. Crone, L. Haas, S. Malaika, T. Mukai, D. Wolfson, and C. Baru. Data access and management services on grid. http://www.cs.man.ac.uk/grid-db/documents.html.

[15] J. Saltz and et.al. Driving scientific applications by data in distributed environments. In *Dynamic Data Driven Application Systems Workshop, held jointly with ICCS 2003*, Melbourne, Australia, June 2003.

[16] J. Smith, A. Gounaris, P. Watson, N. W. Paton, A. A. Fernandes, and R. Sakellariou. Distributed query processing on the grid. http://www.cs.man.ac.uk/grid-db/documents.html.

[17] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. Passion: Optimized I/O for parallel applications. *IEEE Computer*, 29(6):70–78, June 1996.