Query Planning for Range Queries with User-defined Aggregation on Multi-dimensional Scientific Datasets *

Chialin Chang[†], Tahsin Kurc[†], Alan Sussman[†], Joel Saltz[†+]

† UMIACS and Dept. of Computer Science
University of Maryland
College Park, MD 20742

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

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

Abstract

Applications that make use of very large scientific datasets have become an increasingly important subset of scientific applications. In these applications, the datasets are often multi-dimensional, i.e., data items are associated with points in a multi-dimensional attribute space. The processing is usually highly stylized, with the basic processing steps consisting of (1) retrieval of a subset of all available data in the input dataset via a range query, (2) projection of each input data item to one or more output data items, and (3) some form of aggregation of all the input data items that project to the each output data item. We have developed an infrastructure, called the *Active Data Repository* (ADR), that integrates storage, retrieval and processing of multi-dimensional datasets on shared-nothing architectures. In this paper we address query planning and execution strategies for range queries with user-defined processing. We evaluate three potential query planning strategies within the ADR framework under several application scenarios, and present experimental results on the performance of the strategies on a multiprocessor IBM SP2.

1 Introduction

Large amounts of data are being generated in many scientific and engineering studies by detailed simulations, and by sensors attached to devices such as satellites and microscopes. Hence, storage, retrieval, processing and analyzing very large amounts of scientific data has become an important part of scientific research. Typical

*This research was supported by the National Science Foundation under Grants # ACI-9619020 (UC Subcontract # 10152408) and #BIR 9318183, and the Office of Naval Research under Grant #N6600197C8534. The Maryland IBM SP2 used for the experiments was provided by NSF CISE Institutional Infrastructure Award #CDA9401151 and a grant from IBM.

examples of very large scientific 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 [13]), simulation of a flame sweeping through a volume [18], archives of raw and processed remote sensing data (e.g. AVHRR [17]), and archives of medical images (e.g. high resolution confocal light microscopy, MRI). 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, 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 more than one thousand slides per day.

Scientific applications that make use of large datasets have several important characteristics. The datasets are usually *multi-dimensional*, i.e., each data item in a dataset is associated with a point in a multi-dimensional *attribute space* defined by several attributes of the data item. Applications usually make use of a subset of all possible data available in the dataset. In satellite data processing, for instance, an application may only use sensor data in a limited spatio-temporal region and from a subset of sensor bands. Applications may also combine existing attributes to synthesize new attributes. Since the same physical entity may be described in a complementary manner by different types of datasets, applications may need to generate new datasets by performing joins over pre-existing datasets. Our study of a large set of applications [1, 2, 6, 11, 14, 18, 21] indicates that the processing is often highly stylized and shares several important characteristics. The basic processing step usually consists of mapping multi-dimensional coordinates of the retrieved data items to the coordinates of the proper output data items, and computing output data items by some form of (associative and commutative) aggregation operation over all the retrieved input items mapped to the same output data item.

In a database management system (e.g., an object-relational DBMS), the access and processing patterns of these applications can be represented as multi-dimensional range queries with user-defined aggregation operations. However, efficient execution of such range queries is a challenging task. Storing very large amounts of data requires use of disk farms, composed of distributed collections of disks, thereby requiring an effective distribution of the data items across all the disks in the system. Moreover, user-defined functions in the queries may involve expensive operations, which can benefit greatly from parallel processing. In addition, as aggregation functions usually result in significant reductions in data volume, it is desirable to perform the computation where the data is stored, so that shipping large volumes of data across a network can be avoided. However, despite the reduction in data size, output datasets can still be very large, and may not even fit into the available memory in the parallel database system.

In this paper we address query planning for efficient execution of range queries with user-defined functions on shared-nothing architectures. In particular, we target the problems of partitioning the computational workload across processors, minimization of I/O and communication overheads, and handling of out-of-core output datasets during query processing. We have developed the *Active Data Repository* (ADR) [4, 5], an infrastructure for building parallel databases that enables integration of storage, retrieval and processing of multi-dimensional scientific datasets on a parallel machine. ADR is designed to make it possible to carry out aggregation operations efficiently on shared-nothing architectures. In this paper we evaluate three potential query planning strategies within the ADR framework, under several controlled application scenarios. The goal of this study is to develop methods and cost models based on the characteristics of applications and their associated datasets, along with the machine configuration the system is running on, to guide (and automate) selection of appropriate query planning strategies. This paper presents initial experimental results on the performance of the strategies on a multiprocessor IBM SP.

2 Overview

2.1 Range Queries on Multi-Dimensional Scientific Datasets

In this work we target range queries with user-defined processing that have the following components:

- A reference to an input dataset **I** and an output dataset **O**. Both input datasets and output datasets have underlying multi-dimensional attribute spaces. That is, each data item is associated with a point in a multi-dimensional attribute space defined by the attributes of the data item. The data dimensions can be spatial coordinates, time, or varying experimental conditions such as temperature or velocity. Data items may be sparsely and/or irregularly distributed in the underlying attribute space.

- A multi-dimensional bounding box, referred to here also as a *query window*, defined in the attribute space of **I** or **O**. The bounding box defines the region of interest in a multi-dimensional attribute space.

- A reference to user-defined `projection` and `aggregation` functions. A `projection` function maps an element in **I** to one or more elements in **O**. The mapping is from an element's coordinates in the multi-dimensional attribute space underlying **I** to coordinates in the attribute space of **O**. An `aggregation` function describes how data items in **I** that map to the same data item in **O** are aggregated. The aggregation functions allowed correspond to the *distributive* and *algebraic* aggregation functions defined by Gray et. al [12]. That is, the correctness of the result does not depend on the order the input data items are aggregated.

- A specification of how the final output dataset is handled. There are three possibilities: (1) The output is consumed by the application that performed the range query. (2) The output can be inserted into the database as a new dataset. In this case, **O** corresponds to the dataset to be added. (3) An existing dataset can be updated. The reference to dataset **O** corresponds to the dataset in the database to be updated. In this case, aggregated input data items are combined with the corresponding previously existing output data items, and the results are written back to the database, as for (2). The only difference from (2) is that the output data items may be initialized with their values from the database, instead of with the identity element for the aggregation function.

Typical examples of applications that make use of multi-dimensional scientific datasets are satellite data processing applications [1, 21, 6], the Virtual Microscope and analysis of microscopy data [2, 11], 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 (longitude and latitude) and the time the reading was recorded. In a typical analysis [1, 21], a range query defines a bounding box that covers a part or all of the surface of the earth over a period of time. Data 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. Composite images can be added to the database as new datasets. In addition, new sensor readings can be combined to modify the images already existing in the database (e.g., better sensor values can replace previously selected sensor values in a computed image). Another example is the *Virtual Microscope* [2, 11], which supports the ability to interactively view and process digitized data arising 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 mapping onto a single grid point, to avoid introducing spurious artifacts into the displayed image.

2.2 The Active Data Repository

In this section we briefly describe the *Active Data Repository* (ADR) [4, 5]. ADR consists of a front-end and a back-end. The front-end interacts with client applications and relays the range queries issued by the clients to the back-end, which consists of a set of processing nodes and multiple disks attached to these nodes. During query processing, the back-end nodes are responsible for retrieving the input data and performing `projection` and `aggregation` operations over the data items retrieved to generate the output products as defined by the range query. ADR has been developed as a set of modular services in C++. Through use of these services, ADR provides support for common database operations, including index lookup, data retrieval, memory management, and scheduling of processing across a parallel machine, while at the same time allowing for customization for different types of processing over widely varying datasets with underlying multi-dimensional attribute spaces.

We now describe how datasets are stored in ADR, and describe the ADR query planning and query execution services in more detail.

2.3 Storing Datasets

In order to achieve low latency retrieval of data, ADR datasets are partitioned into chunks, each of which consists of one or more data items from the same dataset. A chunk is the unit of I/O and communication operations in the ADR. That is, a chunk is retrieved as a whole during query processing. Since each data item is associated with a point in a multi-dimensional attribute space, a chunk is associated with a minimum bounding rectangle (MBR) that encompasses the coordinates of all the items in the chunk. An index is constructed on the MBRs of the chunks, and the index is used to find the chunks that intersect a query window during query processing. Chunks are declustered across the disks using a declustering algorithm [8, 9, 15] to achieve I/O parallelism during query processing. Each chunk is assigned to a single disk, and is read and/or written during query processing only by the local processor to which the disk is attached. If a chunk is required for processing by one or more remote processors, the chunk is sent to those processors by the local processor via interprocessor communication.

2.4 Query Planning

The task of the query planning service is to determine a query plan to efficiently process a range query based on the amount of available resources in the back-end. A query plan specifies how parts of the final output are computed and the order the input data chunks are retrieved for processing. In order to hold partial results generated during aggregation operations, an intermediate data structure, referred to as an *accumulator*, may be required. The partial results computed into the accumulator can be further processed in the final phase of query execution to produce the final output dataset. In the rest of the paper, we assume that an accumulator is always used during query processing, and that the accumulator is partitioned into chunks, each of which corresponds to a chunk in the output dataset, since all existing ADR applications require an accumulator.

Query planning is carried out in two steps; *tiling* and *workload partitioning*. In the tiling step, if the size of the accumulator is too large to fit into the back-end main memory, the accumulator is partitioned into *tiles*, each of which contains a distinct set of one or more accumulator chunks, so that each tile fits entirely into memory. Indices, provided by the ADR indexing service, are then used to locate the data chunks that must be retrieved from the disks by each back-end process for each tile. Since a projection function may map an input element to multiple output elements, an input chunk may also be mapped to multiple accumulator chunks. In other words, a projection function determines how the input data chunks are distributed in the output dataset attribute space. An input chunk needs to be retrieved multiple times if the output chunks it maps to are assigned to different tiles. In the workload partitioning step, the workload associated with each tile (i.e., processing for 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. In Section 3 we present query planning strategies that implement different tiling and workload partitioning schemes.

2.5 Query Execution

The query execution service manages all the resources in the system and carries out the query plan generated by the query planning service. The primary feature of the query execution service is its ability to integrate data retrieval and processing for a wide variety of applications. This is achieved by pushing processing operations into the storage manager and allowing processing operations to access the buffer used to hold data arriving from disk. As a result, the system avoids one or more levels of copying that would be needed in a layered architecture where the storage manager and the processing belonged to different layers.

To further reduce query execution time, the query execution service overlaps disk operations, network operations and processing as much as possible. It does this by maintaining explicit queues for each kind of operation (data retrieval, message sends and receives, data processing) and switches between them as required. Pending asynchronous I/O and communication operations left in the operation queues are polled and, upon their completion, new asynchronous functions are initiated when more work is expected and buffer space is available. Data chunks are therefore retrieved and processed in a pipelined fashion. The processing of a query on a back-end processor progresses through the following phases for each tile:

1. **Initialization**. Accumulator chunks in the current tile are allocated space in memory and initialized. If an existing output dataset is required to initialize accumulator elements, an output chunk is retrieved by the processor that has the chunk on its local disk, and the chunk is forwarded to the processors that require it.

2. **Local Reduction**. Input data chunks on the local disks of each back-end processor are retrieved and aggregated into the accumulator chunks 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 chunks for the current tile are computed from the corresponding accumulator chunks computed in phase 3. If the query creates a new dataset, output chunks are declustered across the available disks, and each output chunk is written to the assigned disk. If the query updates an already existing dataset, the updated output chunks are written back to their original locations on the disks.

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.

3 Query Planning Strategies

In this section we describe three query planning strategies that use different workload partitioning and tiling schemes. To simplify the presentation, we assume that the target range query involves only one input and one output dataset. Both the input and output datasets are assumed to be already partitioned into chunks and declustered across the disks in the system. In the following discussions we assume that an accumulator chunk is allocated in the memory for each output chunk to hold the partial results, and that the total size of the accumulator exceeds the aggregate memory capacity of the parallel machine, so that tiling is needed.

We define a *local input/output chunk* on a processor as an input/output chunk stored on one of the disks attached to that processor. Otherwise, it is a *remote* chunk. A processor *owns* an input or output chunk if it is a local input or output chunk. A *ghost chunk* (or *ghost cell*) is an accumulator chunk allocated in the memory of a processor that does not own the corresponding output chunk.

For query planning, one of the back-end nodes is designated as the master node, which generates the query plan and broadcasts it to all other back-end nodes. Prior to query planning, the local input and output chunks that intersect the query window are determined in each back-end node, including the master node. Each back-end node sends the minimum bounding rectangles of its input and output chunks to the master node to use in query planning. At the end of the tiling and workload partitioning step of query planning, the master processor sends the chunk assignments and workload partitioning information for each tile to the processors that need the information. Each processor uses this information to compute how many chunks (input, output, accumulator, ghost chunk) to allocate, retrieve, process, and communicate in the various phases of query execution.

3.1 Fully Replicated Accumulator (FRA) Strategy

In this scheme each processor is assigned the responsibility to carry out processing associated with its local input chunks. The accumulator is partitioned into tiles, each of which fits into the local memory of a single back-end processor. Figure 1 shows the tiling and workload partitioning step for this strategy. When an output chunk is assigned to a tile, the corresponding accumulator chunk is put into the set of local accumulator chunks in the processor that owns the output chunk, and is assigned as a ghost cell on all other processors. This scheme effectively replicates all of the accumulator chunks in a tile on each processor, and each processor generate partial results using its local input chunks. These partial results are combined into the final result in the *global combine* phase of query execution. Ghost chunks are forwarded to the processors that own the corresponding output (accumulator) chunks to produce the final output product. Since a projection function may map an input chunk to multiple output chunks, an input chunk must be retrieved multiple times if the corresponding output chunks are assigned to different tiles. In the implementation of all the query strategies described in this paper, we use a *Hilbert space-filling curve* [15, 9, 10] to order the output chunks. Our goal is to minimize the total length of the boundaries of the tiles, by assigning spatially close chunks in the multi-dimensional attribute space to the same tile, to reduce the number of input chunks crossing one or more boundaries. The advantage of using Hilbert curves is that they have good clustering properties [15], since they preserve locality. In our implementation, the mid-point of the bounding box of each output chunk is used to generate a Hilbert curve index. The chunks are sorted with respect to this index, and selected in this order for tiling (step 4 in Figure 1). The current implementation, however, does not take into account the distribution

```
1.  set Memory = Minimum of the size of memory in each processor (for holding accumulator chunks)
2.  set Tile = 1, MemoryUsed = 0
3.  while (there is an unassigned output chunk)
4.      Select an output chunk
5.      set ChunkSize = Size of the corresponding accumulator chunk
6.      if ((ChunkSize+MemoryUsed) > Memory)
7.          Tile = Tile + 1
8.          MemoryUsed = ChunkSize
9.      else
10.         MemoryUsed = MemoryUsed + ChunkSize
11.     Assign the output (accumulator) chunk to tile Tile
        (* Workload partitioning step *)
12.     Let k be the processor that owns the output chunk
13.       Add the accumulator chunk to the set of local accumulator
          chunks of k for this tile
14.       Add the accumulator chunk to the set of ghost cells on all
          other processors for this tile
15.       Add set of local input chunks of k that map to this output chunk to
          set of input chunks to be retrieved by k during query execution for this tile
16. set NumberOfTiles = Tile
```

Figure 1: Tiling and workload partitioning for the fully replicated accumulator strategy.

of input chunks in the output attribute space, so for some distributions of the input data in its attribute space there can still be many input chunks intersecting multiple tiles, despite a small boundary length.

Executing step 15 of the while loop requires either an efficient *inverse projection* function or an efficient search method, either of which must return the input chunks that map to a given output chunk. In some cases it may be less expensive to find the projected output chunks for each input chunk. For example, the input chunks may be irregularly and sparsely distributed in the input attribute space, while the output may be a dense and regular array such as a raster image, with the output chunks as regular subregions of the array. In such cases, step 15 can be carried out in a separate loop, iterating over input chunks, finding the output chunks they map to, and adding the input chunks to the appropriate tiles. To run step 15 as a separate loop, the implementation must store the assigned tile number with each output chunk.

3.2 Sparsely Replicated Accumulator (SRA) Strategy

The fully replicated accumulator strategy eliminates interprocessor communication for input chunks, by replicating all accumulator chunks. However, this is wasteful of memory, because the strategy replicates each accumulator chunk in every processor even if no input chunks will be aggregated into the accumulator chunks in some processors. This results in unnecessary initialization overhead in the *initialization* phase of query execution, and extra communication and computation in the *global combine* phase. The available memory in

1. **for** (each processor `i`)
   set `Memory(i)` = Size of memory in `i` (for holding accumulator chunks)
2. **set** `Tile` = 1, `MemoryFull` = 0
3. **while** (there is an unassigned output chunk)
4. Select an output chunk
5. Let S_o be the set of processors having at least one input chunk
   that projects to this output chunk
6. **set** `ChunkSize` = Size of the corresponding accumulator chunk
7. **for** (`p` **in** S_o)
8. **if** ((`Memory(p)` $-$`ChunkSize`) < 0) **set** `MemoryFull` = 1
9. **if** (`MemoryFull` == 1)
10. `Tile` = `Tile` + 1
11. **for** (`p` **in** S_o) **set** `Memory(p)` = (size of memory on `p`) $-$ `ChunkSize`
12. **for** (`p` **not in** S_o) **set** `Memory(p)` = size of memory on `p`
13. **set** `MemoryFull` = 0
14. **else**
15. **for** (`p` **in** S_o) **set** `Memory(p)` = `Memory(p)` $-$ `ChunkSize`
16. Assign the output (accumulator) chunk to tile `Tile`
    (* **Workload partitioning step** *)
17. Let `k` be the processor that owns the output chunk
18. Add the accumulator chunk to the set of local accumulator
    chunks of `k` for this tile
19. Add the accumulator chunk to the set of ghost cells on each
    processor in S_o for this tile
20. Add set of local input chunks of `k` that map to this output chunk to
    set of input chunks to be retrieved by `k` during query execution
21. **set** `NumberOfTiles` = `Tile`

Figure 2: The tiling and workload partitioning step for the sparsely replicated accumulator strategy.

the system also is not efficiently employed, because of unnecessary replication. Such replication may result in more tiles being created than necessary, which may cause a large number of input chunks to be retrieved from disk more than once. The tiling step of the sparsely replicated accumulator strategy is shown in Figure 2.

In this strategy, a ghost chunk is allocated only on processors owning at least one input chunk that projects to the corresponding accumulator chunk. Replicating accumulator chunks sparsely in this way requires that we find the corresponding set of processors for each output chunk during the tiling step (step 5 in Figure 2). As was stated in the previous section, sometimes it may be easier to find the projected output chunks for each input chunk. For those cases, an alternative solution is to maintain a list for each output chunk to store the set of processors that require allocating an accumulator chunk. The list is created prior to the tiling step by iterating over the input chunks, projecting them to output chunks, and storing the result (processor id) with each output chunk.

```
1.  for (each processor i)
2.    set Memory(i) = Size of memory in i (for holding accumulator chunks)
3.    set Tile(i) = 1
4.  while (there is unassigned output chunk)
5.     Select an output chunk
6.     Let p be the processor that owns this output chunk
7.     set ChunkSize = Size of the corresponding accumulator chunk
8.     if ((Memory(p) −ChunkSize) < 0)
9.        set Tile(p) = Tile(p) + 1
10.       set Memory(p) = (size of memory on p) − ChunkSize
11.    else
12.       set Memory(p) = Memory(p) − ChunkSize
13.    Assign the output (accumulator) chunk to tile Tile(p)
       (* Workload partitioning step *)
14.      Add accumulator chunk to the set of local accumulator
         chunks of p for this tile
15.      Add all the local and remote input chunks that map to the output chunk
         to the set of input chunks to be retrieved and processed by p for this tile
16. set NumberOfTiles = maximum of Tile(p)
```

Figure 3: The tiling and workload partitioning step for the distributed accumulator strategy.

3.3 Distributed Accumulator (DA) Strategy

In this scheme the output (accumulator) chunks in each tile are partitioned into disjoint sets, referred to as *working sets*. Each processor is given the responsibility to carry out the operations associated with the output chunks in a working set. The tiling and workload partitioning step is shown in Figure 3. Output chunks are selected (step 5) in Hilbert curve order, as for the other two schemes.

Local chunks for each processor are assigned to the same tile until the memory space allocated for the accumulator on that processor is filled. Since accumulator chunks are not replicated on other processors, no ghost chunks are allocated. Therefore this scheme minimizes the use of the memory for accumulators (and output chunks) in the system. In the other schemes, local memory space is the factor limiting the number of output chunks assigned to a tile. Since the distributed accumulator strategy can assign more chunks to a tile, it produces fewer tiles than the other schemes. Hence, the number of input chunks that must be retrieved for more than one tile is likely to be less than in the other schemes.

In the algorithm shown in Figure 3, the working set of a processor for a tile is composed of only the local output chunks in that processor. This strategy avoids interprocessor communication during query execution for accumulator chunks and ghost chunks. In the other schemes, output chunks may need to be communicated in the *initialization* phase of query execution to initialize ghost chunks. Similarly, in the *global combine* phase, ghost cells in each processor must be transmitted to the processors that own the local accumulator chunks. On the other hand, the distributed accumulator strategy introduces communication in the *local reduction* phase for input chunks; all the remote input chunks that map to the same output chunk must be forwarded to the processor that owns the output chunk. Since a projection function may map an input chunk to multiple output chunks, an input chunk may be forwarded to multiple processors. In addition, a good declustering strategy could cause almost all input chunks to be forwarded to other processors, because an input chunk and the output chunk(s) that it projects to are unlikely to be assigned to the same processor. These tradeoffs will be evaluated for several application scenarios in Section 4.

4 Experimental Results

In this section we present an experimental performance evaluation of the three query execution strategies on a 16 node IBM SP2. Each node of the IBM SP2 is a thin node with 128 MB of memory; nodes are interconnected with the High Performance Switch, which provides 40MB/sec peak communication bandwidth. In these experiments data was stored on one disk per node.

A complete evaluation of the strategies requires a detailed sensitivity analysis over a large set of parameters, including the number of input and accumulator chunks, the number of data items and the extent in the multi-dimensional attribute space of each chunk, the distribution of the bounding rectangles for the input and accumulator chunks in their corresponding attribute spaces, and the time it takes to read, communicate, initialize and process the chunks. We present the results of an initial set of experiments, in which we vary the extents and the distribution of the bounding rectangles for the input chunks, while keeping the other parameters constant. Although we are not varying the distribution and extents of the accumulator chunks, we can still emulate different mappings (projection functions) between the input and accumulator chunks, by varying the distribution and extents of the input chunks. To further limit the parameter space, we fix the computation time for initializing and processing the chunks at a constant value. We are in the process of performing more detailed experiments with variable chunk sizes and computation costs, to investigate their effects on the performance of the three strategies. Since our main focus is to evaluate the query execution performance of the query planning strategies, all the execution times do not include the time to perform query planning.

We employ synthetic input and output datasets in our experiments to evaluate the strategies under controlled scenarios. The output dataset has an underlying 2D attribute space of integer values (e.g., spatial coordinates) bounded by a rectangle of 10K$\times$10K. The entire output attribute space is regularly partitioned into non-overlapping rectangles, with each rectangle representing an accumulator chunk in the output dataset.

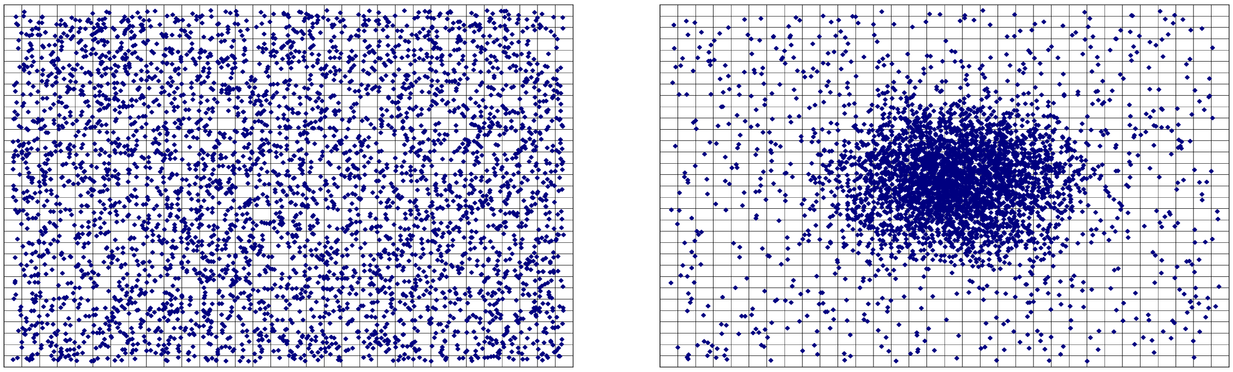

Figure 4: Datasets with uniform (left) and skewed (right) distributions. The dots in each figure show the mid-points of the bounding rectangles of input chunks after projection into the 2D output attribute space.

The input datasets used in the experiments have a 3D attribute space of integer values, bounded by a box of 10K×10K×10K. We used a simple `projection` function that projects a point from the 3D input space into the 2D output space by discarding the coordinate in the third dimension. To emulate different application scenarios, the input chunks were placed into the 3D input attribute space using two different distributions; *uniform* and *skewed*. The skewed distribution was generated by placing the mid-points of 20% of the bounding rectangles of all input chunks using uniform distribution, and placing the mid-points of the remaining bounding rectangles using a normal distribution (with a mean of 5120 and a standard deviation of 1024), in all 3 dimensions of the attribute space. Figure 4 shows the distribution of the mid-points of the bounding rectangles for the input chunks using the two distributions. The skewed distribution places more input chunks towards the center of the input space. Since we are projecting the input chunks into the output space by discarding the third dimension of the input space, the accumulator chunks near the center of the output space would therefore intersect with more input chunks than those located away from the center. This distribution represents a class of applications where the distribution of input chunks in the output space is localized and therefore requires different amounts of processing across the accumulator chunks. Typical examples include applications using adaptive mesh refinement algorithms, studies conducted by earth scientists that employ remote-sensing data at different resolutions, and medical analysis applied to microscopy images taken at different magnifications.

For the experiments, we partitioned the output attribute space into 32×32 accumulator chunks, each with a minimum bounding rectangle of 320×320. The number of input chunks was fixed at 4096 for both distributions. Both input and accumulator chunks are 128KB in size, and were declustered across the 16 disks in the system using Hilbert curve based declustering algorithms [9]. The result is that all disks are assigned approximately the same number of input (accumulator) chunks, and chunks that are spatially close in the respective attribute spaces are distributed to different disks. We used a single query window that covers the entire input and output attribute spaces. The computation in each phase of the query execution is simulated by a constant delay. Our experience with scientific applications indicate that the processing in the *initialization* and *global combine* phases is usually much less expensive than that in the *local reduction* phase. To model those costs, we used a delay of 1 millisecond for both initializing an accumulator chunk in the initialization phase and combining the partial results for an accumulator chunk in the global combine phase. In selecting the cost for the local reduction phase computation, we want to choose a value that is significantly larger than the computation costs in the initialization and global combine phases, but also is small enough so that the cost relative to communication and I/O in the queries is not too small. Otherwise, the queries become very computationally intensive, and the performance of the queries under all the planning strategies is affected only by the computational load balance across the processors. For each intersecting (input chunk, accumulator chunk) pair we used a delay of 10 milliseconds to emulate the processing in the local reduction phase. Therefore, an input chunk that maps to a larger number of accumulator chunks takes more time to process. Similarly, an accumulator chunk to which a larger number of input chunks map takes longer to process.

We first investigate how the amount of memory allocated for the accumulator and for the I/O and communication buffers affects the performance of the three strategies. In this set of experiments, the total amount of memory is fixed at 64 MB per processor, and the amount of memory allocated for accumulators and for I/O and communication buffers is varied. We use input chunks with extent 320×320 and 1280×1280, distributed using both uniform and skewed distributions. Figure 5 shows the query execution times for the three strategies; *full replicated accumulator* (FRA), *sparse replicated accumulator* (SRA), and *distributed accumulator* (DA), using 4 MB, 9 MB, 16 MB, 48 MB and 60 MB of memory per processor for the accumulators. DA generates 3 tiles with 4 MB and 1 tile with 9 MB or more, while FRA and SRA generate 34 tiles and 33 tiles at 4 MB, respectively, and 3 tiles and 1 tile at 60 MB. This is because DA makes more efficient use of the aggregated memory in the system. As is shown in the figure, the performance of all the strategies increases as the memory allocated for accumulators increases, but flattens out at about 32 MB. The performance improvement is mainly because decreasing the number of tiles decreases total I/O volume. For example, using the FRA scheme with a uniform data distribution, the volume of I/O for a processor is 60 MB and 47 MB, for accumulator memory sizes of 4 MB and 60 MB, respectively. However, since the I/O time is small and part of the I/O time overlaps with computation, the decrease in overall execution time is not large. We observe that the performance deteriorates at 60 MB. We suspect that this is due to excessive page faults

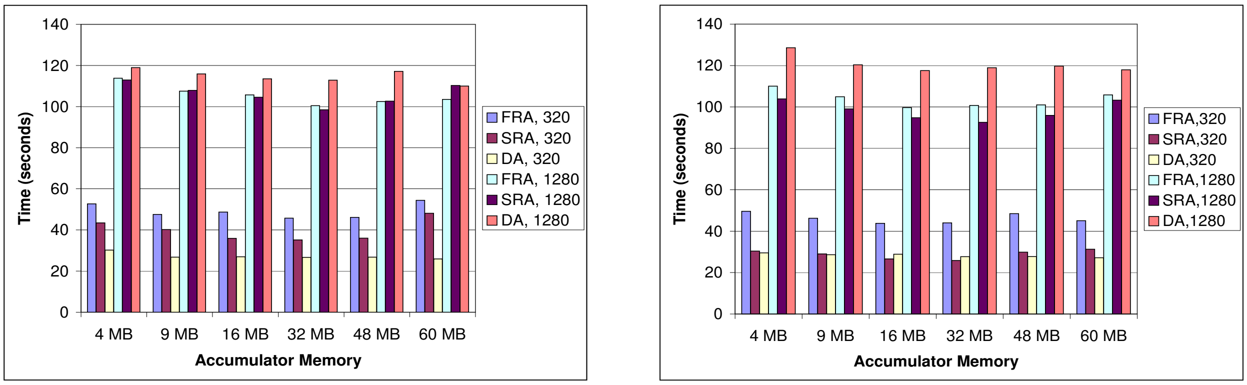

Figure 5: Total query execution time for input datasets of uniform (left) and skewed (right) distributions, using different memory configurations.

for keeping an accumulator tile of 60 MB in memory, but a complete analysis requires further investigation. Although at least 4 MB of memory was allocated for I/O and communication buffers, less memory was required at run-time to keep the processors busy. Since the best performance is achieved with accumulator memory set to 32 MB, we will use this memory configuration for the rest of the experiments presented.

Figure 6 shows query processing time as the extent of the input bounding rectangles in each dimension is varied. The x-axis in the figures represents the length of each dimension of the query bounding rectangle. Varying the extent of bounding rectangles varies the number of accumulator chunks an input chunk maps to. As expected, enlarging the bounding rectangle of an input chunk increases the number of input chunks that each accumulator chunk must process, hence increases total processing time. This can be seen from the increase in the time for the local reduction phase of the query, in the execution time breakdowns shown in Figure 7. Figure 6 also shows that DA performs better than, or as well as, the other strategies when the input chunk size is small, but performs worse when the input chunk becomes large. This is mainly because the communication overhead in DA is proportional to the average *fanout* of an input chunk, where fanout is the number of accumulator chunks an input chunk maps to. As a result of Hilbert curve based declustering algorithm, the set of accumulator chunks to which a given input chunk maps is likely to be distributed over multiple processors. Thus, for larger input chunks, an input chunk intersects more accumulator chunks, and the input chunk needs to be forwarded to more processors. The volume of communication in FRA and SRA, on the other hand, is proportional to the number of accumulator chunks. In FRA, all processors always broadcast their local accumulator chunks to other processors regardless of the size of the input chunk extent. Therefore, with a small input chunk extent, where DA sends each input chunk to a relatively small number of processors, FRA incurs more overhead due to interprocessor communication. SRA, however, generates

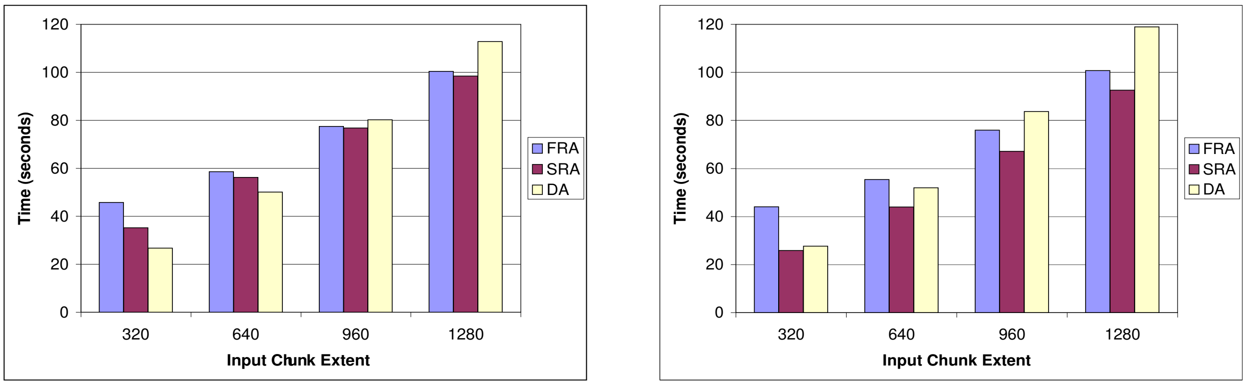

Figure 6: Total query execution time for input datasets of uniform (left) and skewed (right) distributions with different input bounding rectangle extents.

less communication volume than FRA, since it avoids unnecessary replication. The difference between the volumes of interprocessor communication that occur in the DA and FRA schemes can be computed from the following formula:

$$\frac{N}{P} \times f \times B - \frac{(P-1)M}{P} \times 2 \times S \qquad (1)$$

where N is the total number of input chunks, f is the average fanout of the input chunks, B is the size of an input chunk, P is the number of processors, M is the total number of accumulator chunks, and S is the size of an accumulator chunk. The constant 2 comes from the fact that in FRA each accumulator chunk is communicated twice, once during the initialization phase and once during the global combine phase of query execution. With the experimental parameters we are using, Equation (1) is greater than zero when f is greater than 8, which occurs when the extent of the input chunks exceeds 640. Thus, as can be seen from Figure 6, the DA scheme performs worse than the FRA and SRA schemes when the extent of the input chunks is larger than 640. Note that the average fanout of input chunks can be computed during query planning.

Figure 7 shows the breakdown of the total execution time into phases. The figure shows that the output handling phase (Output), in which each processor writes updated output chunks back to disk, takes a very small amount of time since each processor writes only a small number of output chunks. Also, with small input chunks (i.e., small bounding rectangles), the times spent in the initialization, reduction and combine phases are approximately the same. However, the reduction phase becomes dominant with larger input chunks.

Figure 8 shows how much time is spent in I/O, communication, computation and waiting during the execution of the query. The figure shows that communication and computation times are the dominant

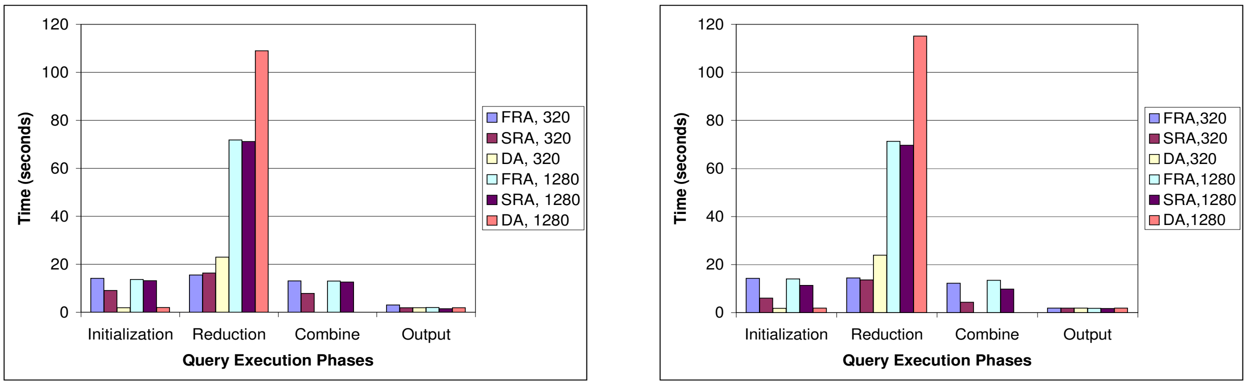

Figure 7: Execution time breakdowns by phases for uniform (left) and skewed (right) distributions.

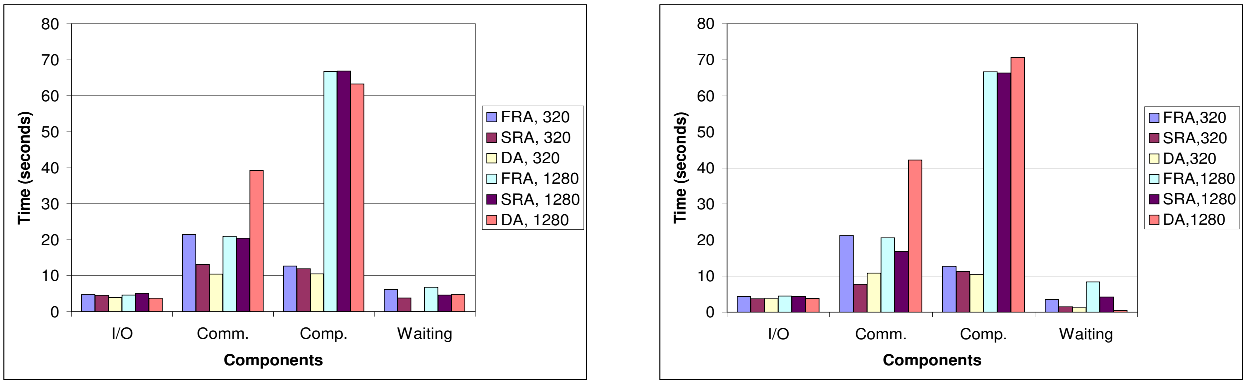

Figure 8: Execution time breakdowns by components for uniform (left) and skewed (right) distributions.

components. The waiting time is the total time (over all tiles for all phases) processors spend idle at the end of the various phases during query execution. The waiting time effectively measures the load imbalance across the processors due to computation, I/O and interprocessor communication.

In these experiments, query planning takes very little time to generate the required query plans. For example, with an input chunk extent of 320, it takes 0.82, 0.93 and 0.79 seconds to generate a query plan for FRA, SRA and DA, respectively. An additional 0.2 seconds is used for communication between the master node and the other nodes. With an input chunk extent of 1280, where input chunks are mapped to more accumulator chunks, it takes 1.1, 1.3 and 1.2 seconds to generate a query plan for each scheme, and 0.3 seconds for communication.

5 Related Work

Parallel database systems have been a major topic in the database community [7] for a long time, and much attention has been devoted to the implementation and scheduling of parallel joins [16, 19]. As in many parallel join algorithms, our query strategies exploit parallelism by effectively partitioning the data and workload among the processors. However, the characteristics of the distributive and algebraic aggregation functions allowed in our queries enable us to deploy more flexible workload partitioning schemes through the use of ghost chunks. Several extensible database systems have been proposed to provide support for user-defined functions [3, 22]. The incorporation of user-defined functions into a computation model as general as the relational model, can make query optimization very difficult, and has recently attracted much attention [20]. Our system, on the other hand, implements a more restrictive processing structure that mirrors the processing of our target applications. Good performance is achieved through effective workload partitioning and careful scheduling of the operations to obtain good utilization of the system resources, not by rearranging the algebraic operators in a relational query tree, as is done in relational database systems.

6 Conclusions and Future Work

In this paper we have addressed query planning strategies to enable efficient execution of range queries with user-defined aggregation functions, targeting multi-dimensional scientific datasets. We have evaluated three potential query planning strategies that employ different tiling and workload partitioning schemes within the ADR framework.

Our results indicate that no one scheme is always best. The relative performance of the various query planning strategies changes with the application characteristics. The strategies presented in this paper represent two extreme approaches. The full and sparse replicated accumulator strategies lie one end of the spectrum of possible strategies: processing is performed on the processors where input chunks are stored. The distributed accumulator strategy, on the other hand, lies at the other end: processing is carried out on the processors where output blocks are stored. Our experimental results suggest that a hybrid strategy may be the best approach. The tiling and workload partitioning steps can be formulated as a multi-graph partitioning problem, with input and output chunks representing the graph vertices, and the mapping between input and output chunks provided by the *projection* function representing the graph edges. A planning algorithm based on graph partitioning could provide an effective means for implementing a general hybrid strategy, and we are currently investigating this line of work.

One of the long-term goals of our work on query planning strategies is to develop simple but reasonably accurate cost models to guide and automate the selection of an appropriate strategy. In this paper, we showed a very simple formula (Eq. 1) that seems able to predict the relative performance of the various strategies under changing application characteristics. This is encouraging since it shows that simple cost models may perform well for some application scenarios. However, answering the question "under what circumstances do the simple cost models provide accurate or inaccurate results?", and "how can we refine the cost model in situations where it does not provide reasonably accurate results?" require further research. We plan to evaluate our strategies on additional applications and on other parallel architectures to further investigate these questions.

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] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. R. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vandenberg. The EXODUS extensible DBMS project: An overview. In D. Zdonik, editor, *Readings on Object-Oriented Database Systems*, pages 474–499. Morgan Kaufman, San Mateo, CA, 1990.

[4] C. Chang, A. Acharya, A. Sussman, and J. Saltz. T2: A customizable parallel database for multi-dimensional data. *ACM SIGMOD Record*, 27(1):58–66, Mar. 1998.

[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 (13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing)*. IEEE Computer Society Press, Apr. 1999. To appear.

[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] 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.

[8] H. C. Du and J. S. Sobolewski. Disk allocation for Cartesian product files on multiple-disk systems. *ACM Trans. Database Syst.*, 7(1):82–101, Mar. 1982.

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

[10] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Philadelphia, PA, Mar. 1989.

[11] 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.

[12] 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 1996 International Conference on Data Engineering*, pages 152–159, Feb. 1996.

[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. To appear.

[14] S. Liang, L. Davis, J. Townshend, R. Chellappa, R. Dubayah, S. Goward, J. JaJa, S. Krishnamachari, N. Roussopoulos, J. Saltz, H. Samet, T. Shock, and M. Srinivasan. Land cover dynamics investigation using parallel computers. In *Proceedings of the 1995 International Geoscience and Remote Sensing Symposium, Quantitative Remote Sensing for Science and Applications.*, pages 332–4, July 1995.

[15] B. Moon and J. H. Saltz. Scalability analysis of declustering methods for multidimensional range queries. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):310–327, March/April 1998.

[16] M. C. Murphay and D. Rotem. Multiprocessor join scheduling. *IEEE Transactions on Knowledge and Data Engineering*, 5(2):322–338, Apr. 1993.

[17] NASA Goddard Distributed Active Archive Center (DAAC). Advanced Very High Resolution Radiometer Global Area Coverage (AVHRR GAC) data. *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] D. Schneider and D. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of the 16th VLDB Conference*, pages 469–480, Melbourne, Australia, Aug. 1990.

[20] P. Seshadri, M. Livny, and R. Ramakrishnan. The case for enhanced abstract data types. In *Proceedings of the 23th VLDB Conference*, Athens, Greece, Aug. 1997.

[21] 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.

[22] M. Stonebraker, L. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, Mar. 1990.

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