Titan: a High-Performance Remote-sensing Database [*]

Chialin Chang, Bongki Moon, Anurag Acharya, Carter Shock,
Alan Sussman, Joel Saltz
Department of Computer Science and UMIACS
University of Maryland, College Park MD 20742
{chialin,bkmoon,acha,ctso,als,saltz}@cs.umd.edu

Abstract

There are two major challenges for a high-performance remote-sensing database. First, it must provide low-latency retrieval of very large volumes of spatio-temporal data. This requires effective declustering and placement of a multi-dimensional dataset onto a large disk farm. Second, the order of magnitude reduction in data-size due to post-processing makes it imperative, from a performance perspective, that the postprocessing be done on the machine that holds the data. This requires careful coordination of computation and data retrieval. This paper describes the design, implementation and evaluation of Titan*, a parallel shared-nothing database designed for handling remote-sensing data. The computational platform for Titan is a 16-processor IBM SP-2 with four fast disks attached to each processor. Titan is currently operational and contains about 24 GB of AVHRR data from the NOAA-7 satellite. The experimental results show that Titan provides good performance for global queries and interactive response times for local queries.*

1. Introduction

Remotely-sensed data acquired from satellite-based sensors is widely used in geographical, meteorological and environmental studies. A typical analysis processes satellite data for ten days to a year and generates one or more images of the area under study. Data volume has been one of the major limiting factors for studies involving remotely-sensed data. Coarse-grained satellite data (4.4 km per pixel) for a global query that spans the shortest period of interest (ten days) is about 4 GB; a finer-grained version of the same data (1.1 km per pixel) is about 65 GB. The output images are usually significantly smaller than the input data. For example, a multi-band full-globe image corresponding to the 4.4 km dataset mentioned above is 228 MB. This data reduction is achieved by composition of information corresponding to different days. Before it can be used for composition, individual data has to be processed for correcting the effects of various distortions including instrument drift and atmospheric effects.

These characteristics present two major challenges for the design and implementation of a high-performance remote-sensing database. First, the database must provide low-latency retrieval of very large volumes of spatio-temporal data from secondary storage. This requires effective declustering and placement of a multi-dimensional dataset onto a suitably configured disk farm. Second, the order of magnitude reduction in data size makes it imperative, from a performance perspective, that correction and composition operations be performed on the same machine that the data is stored on. This requires careful coordination of computation and data retrieval to avoid slowing down either process.

Several database systems have been designed for handling geographic datasets [4, 19, 26, 27]. These systems are capable of handling map-based raster images, map points (e.g. cities) and line segments (e.g. rivers, roads) and provide powerful spatial query operations. They are, however, unsuitable for storing raw remote-sensing data that has not been projected to a map-based coordinate system. Maintaining remote-sensing data in its raw form is necessary for two reasons [23]. First, a significant amount of earth science research is devoted to developing correlations between raw sensor readings at the satellite and various properties of the earth's surface; once the composition operation is performed it is no longer possible to retrieve the original data. Second, the process of generating a map-based image uses a particular projection of a globe onto a two-dimensional grid. Earth scientists use several different projections, each for a different purpose; no single projection is adequate for all uses. In addition, none of these systems have been designed for large disk farms. Given the volume of data retrieved for

[*] This research was supported by the National Science Foundation under Grant #ASC 9318183, NASA under Grant #NAG 11485 (ARPA Project #8874), and ARPA under Grant #DABT 63-94-C-0049 (Caltech Subcontract #9503).

each query, uniprocessor platforms are unable to provide adequate response times.

This paper describes the design, implementation and evaluation of *Titan*, a parallel shared-nothing database designed for handling remote-sensing data. The computational platform for Titan is a 16-processor IBM SP-2 with four IBM Starfire 7200 disks attached to each processor. Titan is currently operational and contains about 24 GB of data from the Advanced Very High Resolution Radiometer (AVHRR) sensor on the NOAA-7 satellite.

The paper focuses on three aspects of the design and implementation of Titan: data placement, query partitioning and coordination of data retrieval, computation and communication. Section 2 provides an overview of the system. Section 3 describes the declustering and data placement techniques used in Titan. The data layout decisions in Titan were motivated by the format of AVHRR data and the common query patterns identified by NASA researchers and our collaborators in the University of Maryland Geography Department. Section 3.2 describes the indexing scheme used by Titan. Section 4 describes the mechanisms used to coordinate data retrieval, computation and interprocessor communication. Section 5 describes the experiments performed to evaluate Titan. Section 6 discusses the results and pinpoints the performance-limiting factors.

We believe our experiences suggest useful guidelines that go beyond remote-sensing databases in their scope. In particular, we expect our techniques for coordinating and balancing computation, communication and I/O are useful for other unconventional databases that need to perform substantial post-processing. Similarly, we believe our results provide additional evidence for the utility of the *minimax* algorithm for declustering multidimensional datasets over large disk farms.

2. System Overview

Titan consists of two parts: (1) a front-end that interacts with querying clients, performs initial query processing and partitions data retrieval and computation; and (2) a back-end that retrieves the data and performs post-processing and composition operations.

The front-end consists of a single host which can be located anywhere on the network. The back-end consists of a set of processing nodes on a dedicated network that store the data and do the computation. The current implementation of Titan uses one node of the 16-processor IBM SP-2 as the front-end and the remaining 15 nodes as the back-end. No data is stored on the disks of the node used as the front-end.

Titan partitions its data set into coarse-grained data blocks and uses a simplified R-tree to index these chunks (see sections 3.1 and 3.2 for details). This index is stored at the front-end which uses it to build a plan for the retrieval and processing of the required data blocks. The size of this index for 24 GB of AVHRR data is 11.6 MB, which is small enough to be held in primary memory.

Titan queries specify four constraints: (1) temporal bounds (a range in *universal coordinated time*), (2) spatial bounds (a quadrilateral on the surface of the globe), (3) sensor type and number, and (4) resolution of the output image. The result of a query is a multi-band image. Each pixel in the result image is generated by composition over all the sensor readings for the corresponding area on the earth's surface.

When the front-end receives a query, it searches the index for all data blocks that intersect with the query. It uses the location information for each block (which is stored in the index) to determine the set of data blocks to be retrieved by each back-end node. In addition, the front-end partitions the output image among all the back-end nodes. Currently, the output image is evenly partitioned by blocks of rows and columns, assigning each back-end node approximately the same number of output pixels. Under this partitioning scheme, data blocks residing on the disks of a node may be processed by other nodes; each back-end node processes the data blocks corresponding to its partition of the output image (see section 4 for details). The front-end distributes the data block requests and output image partitions to all back-end nodes.

Each back-end node computes a schedule for retrieving the blocks from its disks. This schedule tries to balance the needs of all nodes that will process these data blocks. As soon as a data block arrives in primary memory, it is dispatched to all nodes that will process it. Once a data block is available for processing (either retrieved from local disk or forwarded by another node), a simple quadrature scheme is used to search for sensor readings that intersect with the local partition of the output image. After all data blocks have been processed, the output image can either be returned to the front-end for forwarding to the querying client, or it can be stored in a file for later retrieval.

3. Data Placement

Titan addresses the problem of low-latency retrieval of very large volumes of data in three ways. First, it takes advantage of the AVHRR data format and of common query patterns identified by earth science researchers [7, 10, 24], to partition the entire dataset into coarse-grained chunks that achieve good disk bandwidth. Second, it tries to maximize disk parallelism by declustering the set of chunks onto a large disk farm. Finally, it attempts to minimize seek time on individual disks by clustering the chunks assigned to each disk. The following subsections describe each of these techniques used to provide low-latency data retrieval.

3.1. Data Partitioning

AVHRR data is organized as one file per satellite orbit (with 14 orbits per day). Each file consists of a sequence of scan lines, each with 409 pixels. Each pixel consists of five readings, each in a different band of the electro-magnetic spectrum.

The AVHRR files provided by NASA [2] are organized in a band-interleaved form (*i.e.*, all the values for a single pixel are stored consecutively). However, most satellite data processing programs process one of two groups – bands one and two or bands three, four and five [7, 24]. This grouping occurs due to the properties of the bands: the first two bands provide information to estimate the amount of chlorophyll in a region [9] whereas the last three bands can be used to estimate cloud cover and surface temperature [14]. To take advantage of these query patterns, Titan stores AVHRR data in two parts, one containing data for bands one and two and the other containing data for bands three, four and five.

We used three guidelines to determine the size of the basic data block – *i.e.*the unit of data retrieved from disk. First, the size should be large enough to allow efficient retrieval; second, the size should be small enough that most of the data retrieved is useful; and third, the block should be square (square groups of pixels provide better indexing than more elongated groups). From [1] we knew that for our configuration, the best I/O performance is achieved for blocks larger than 128 KB. Therefore, we chose to partition the AVHRR data in tiles of 204x204 pixels. The data blocks containing bands one and two are about 187 KB; the data blocks containing bands three, four and five are about 270 KB. To minimize disk seek time, all data blocks with bands 1 and 2 are stored contiguously on disk, as are all data blocks with bands 3, 4 and 5.

3.2. Indexing Scheme

The Titan index contains spatio-temporal bounds and retrieval keys for the coarse-grained data blocks described in Section 3. For expediency and due to the relatively small number of blocks in the database, we have implemented the index as a simplified R-tree. The index is a binary tree whose interior nodes are bounding quadrilaterals for their children. We use quadrilaterals instead of rectangles to achieve a better fit for spatial bounds. We chose a binary tree for its simplicity; since entire index is held in memory, disk efficiency is not important. Leaf nodes in the index correspond to data blocks and contain spatial and temporal extent, meta-data such as sensor type and satellite number, and the position on disk for each data block. The position of a data block is described by a [disk,offset] pair.

The leaf nodes are arranged in a z-ordering [21] before the index is built. Sorting the leaves spatially allows access to the index as a range tree. Furthermore, it allows interior node keys in the index to better approximate the spatial extent of their children, and reduces the overlap between different interior node keys at the same level in the tree. As a result, searching the index becomes more efficient.

Using a coarse-grained index has several advantages. First, the index supports efficient retrieval of data from disks. Second, the index supports quick winnowing of large portions of the data base when presented with localized queries. Third, the index allows query previews that enable users to quickly refine their queries, without forcing large volumes of data to be retrieved from disks [5]. Finally, the index is extensible – it is easy to include data from other sensors without re-engineering the indexing scheme or re-indexing existing data.

3.3. Declustering

Declustering methods for multidimensional datasets can be classified into two classes: *grid-based* [3, 6] and *graph-theoretic* [15, 16]. Grid-based methods have been developed to decluster Cartesian product files, while graph-theoretic methods are aimed at declustering more general spatial access structures such as grid files [20] and R-trees [11]. A survey of declustering methods can be found in [18].

Since the Titan index is tree-based, we have adopted a graph-theoretic algorithm – Moon *et al.* 's *minimax spanning tree* algorithm [16]. This algorithm was originally proposed for declustering grid files on a large disk farm and has been shown to outperform Fang *et al.* 's Short Spanning Path algorithm [8] for that task. The *minimax* algorithm models the problem as a complete graph – each vertex represents a data block and each edge represents the likelihood that the corresponding data blocks will be accessed together. The key idea of the algorithm is to extend Prim's minimal spanning tree algorithm [22] to construct as many spanning trees as there are disks in the disk farm, and then assign all the blocks associated with a single spanning tree to a single disk. To generate the edge costs, we have chosen the *proximity index* proposed by Kamel and Faloutsos [13]. The alternative we considered, *euclidean distance*, is suitable for point objects that occupy zero area in the attribute space but does not capture the distinction among pairs of *partially overlapped* spatial objects of non-zero area or volume[1].

Prim's algorithm expands a minimal spanning tree by incrementally selecting the minimum cost edge between the vertices already in the tree and the vertices not yet in the tree. This selection policy does not ensure that the increment in the aggregate cost (*i.e.*, the sum of all edge weights inclusive to the group of vertices associated with the spanning tree) due to a newly selected vertex is minimized. Instead,

[1]By partially overlapped objects we mean d-dimensional objects whose projected images on at least any one of d dimensions intersect.

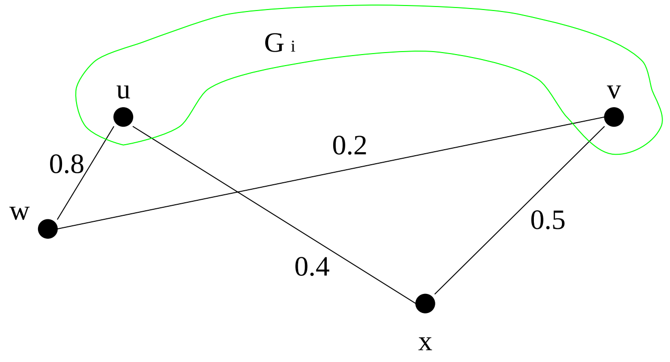

Figure 1. Illustration of expanding spanning trees

the *minimax* algorithm uses a *minimum of maximum costs* policy. For every vertex that has not yet been selected, the algorithm computes a *maximum* of all edge weights between the vertex and the vertices already selected. The selection procedure picks the vertex that has the smallest such value.

For example, in Figure 1, the *minimum of minimum costs* policy will pick up the vertex w and add it to the spanning tree G_i, because the weight of the edge (w, v) is the minimum. However, this decision leads to putting the vertices w and u, which are connected by an edge with a very heavy weight, in the same vertex group represented by the spanning tree G_i. On the other hand, the *minimum of maximum costs* policy will pick up the vertex x and add it to the spanning tree G_i, because the weight of the edge (x, v) is the minimum of maximum costs.

In summary, the *minimax* algorithm (1) seeds M spanning trees (where M is the number of disks) by choosing M vertices instead of choosing a single vertex; (2) expands M spanning trees in round-robin fashion; and (3) uses a *minimum of maximum costs* policy for edge selection, instead of a *minimum of minimum costs* policy. It requires $\mathcal{O}(N^2)$ operations and achieves perfectly balanced partitions (*i.e.*, each disk is assigned at most $\lceil N/M \rceil$ data blocks where M is the number of disks). A detailed description of this algorithm can be found in [16].

3.4. Clustering

In addition to maximizing disk parallelism by declustering, it is important to reduce the number of disk seeks by suitably linearizing the data blocks assigned to a single disk. We have chosen the Short Spanning Path (SSP) algorithm [8] for this purpose. We considered a Hilbert-curve-based scheme as an alternative. Both methods can be used to map multidimensional objects onto a one-dimensional space. It is widely believed that the Hilbert curve achieves the best clustering among space-filling curves [12, 17]. In [16], however, we have empirically shown that the SSP algorithm achieves better declustering (and therefore a better linearization) than a Hilbert-curve-based algorithm.

Finding the shortest spanning path is NP-complete. Therefore, the SSP algorithm employs a heuristic to generate a path that is short, but not necessarily the shortest. The algorithm works by first picking a vertex randomly from a given set of N vertices. Now suppose we have generated a partial path covering k vertices $v_1, \ldots, v_k$, where $k < N$. Then pick another vertex u randomly from the vertex set, and find a position in the path at which the vertex u should be placed by trying the $k + 1$ positions in the path, such that the length of the resulting path is minimized. Once again the *proximity index* is used to measure the distance between vertices.

4. Query Processing

As described in Section 2, Titan consists of a front-end host and a set of back-end nodes. For each query, the front-end uses the index to identify all data blocks that intersect with the spatial and temporal extents of the query. It also partitions the processing for the query among all back-end nodes. Currently, the output image is evenly partitioned by blocks of rows and columns, assigning each back-end node approximately the same number of output pixels. Under this partitioning scheme, data blocks residing on the disks of a node may be processed by other nodes; each back-end node processes the data blocks corresponding to its partition of the output image.

For each data block to be retrieved, the front-end identifies its location (disk number and disk offset) and the identity of all the back-end nodes that need the data contained in the block. In other words, it identifies the *producer* as well as all the *consumers* of the data block. For each back-end node, the front-end computes the number of data blocks it will receive from each of the other back-end nodes. Once all this information is computed, the front-end distributes it to all the back-end nodes.

Each back-end node computes a schedule for retrieving the data blocks on its disks. The primary goal of the schedule is to retrieve data blocks in a balanced manner so as to keep all nodes busy (processing the query). It also tries to avoid disk seeks as far as possible. The scheduling algorithm first assigns a *representative consumer* for each data block. If the block has multiple consumers, the representative-consumer is chosen as follows. If the *producer* of the node is also a consumer, then it is chosen as the representative-consumer; otherwise one of the consumers is chosen randomly. The scheduling algorithm then partitions the list of data blocks into a set of lists, one list for each *[disk, representative-consumer]* pair.

Once a schedule is generated, each back-end node asynchronously executes a loop whose body consists of five

```
while (not all activities are done)
    /* block-read phase */
    issue as many asynchronous disk reads for blocks as possible;
    /* block-send-check phase */
    check all pending block sends, freeing send buffers for completed ones;
    /* block-receive phase */
    check pending block receives, and for each completed one:
        add the receive buffer to the list of buffers that must be processed locally;
        if (more non-local blocks must be obtained) issue another asynchronous receive;
    /* block-read-check phase */
    check pending disk reads;
    for each completed one, generate asynchronous sends to the remote consumers;
    /* block-consume phase */
    if (a block is already available for processing)
        process the block - perform mapping and compositing operations for all readings in the block;
    endif
endwhile
```

Figure 2. Main loop for overlapping computation, I/O and communication.

phases – a block-read phase, a block-send-check phase, a block-receive phase, a block-read-check phase, and a block-consume phase – as shown in Figure 2. Each back-end node also initializes its partition of the output image.

To hide the large latency for I/O accesses and interprocessor communication, the back-end nodes issue multiple asynchronous requests to both the file system and the network interface, so that I/O, communication, and computation may all be overlapped. By keeping track of various pending operations, and issuing more asynchronous operations when necessary, the back-end nodes can move data blocks from their disks to the memory of the consuming back-end nodes in a pipelined fashion. In addition, while I/O and communication are both proceeding, each back-end node can process data blocks as they arrive from either its own local disks or the network.

During the block-read phase, the node issues as many asynchronous reads to the disks as possible. The number of outstanding requests is limited by the number of buffers available and the number of local disks (little or no benefit is gained from too many outstanding reads per disk).

In the block-send-check phase, the node checks for completion of asynchronous sends used to dispatch data blocks to their consumers. When a data block has been delivered to all its consumers, the buffer holding it is released.

In the block-receive phase, the node posts as many asynchronous receives as possible for data blocks that it needs but which are produced by other nodes. The number of outstanding requests is bounded by buffer availability and the number of nodes from which it is expecting to receive data.

In the block-read-check phase, the node checks for the completion of outstanding block-read requests. For each request that has completed, asynchronous sends are generated for all remote consumers, if any.

In the block-consume phase, the node processes a data block that has arrived either over the network or from a local disk. For each block it extracts the data corresponding to its partition of the output image and performs the composition operation. At most one data block is processed within the block-consume phase per iteration. This policy ensures that the outstanding asynchronous operations are polled frequently.

5. Experimental Results

We have evaluated Titan in three ways. First, we have computed static measures that evaluate the quality of the data placement; second, we have conducted simulations to evaluate the performance of data retrieval for a set of queries that cover the globe; and third, we have measured actual performance on the SP-2 for a set of queries that cover land-masses of various sizes.

5.1. Static evaluation of data placement

We used two static measures to evaluate the quality of data placement: (1) the number of k-nearest-neighbor blocks placed on the same disk, for declustering; and (2) the aggregate probability that any pair of adjacent data blocks are fetched together, for clustering. These measures depend only on the data placement and are independent of the distribution, sizes and shapes of queries.

We counted the number of k-nearest-neighbor data blocks assigned to the same disk unit, varying k from 1 to 59 since the total of 55,602 data blocks were distributed over 60 disks. The results are summarized in Table 1. The closer a pair of blocks are to each other, the higher the chance that they are accessed together. Therefore, the reduction of 48 to 70 percent in this measure indicates a potential for substantial performance improvement through declustering.

To estimate the aggregate probability that any pair of adjacent data blocks on the same disk unit are fetched together, we computed the *probability path length* for all disks. This measure is defined as $\sum_{i=1}^{N-1} proximity\_index(Block_i, Block_{i+1})$, where N is

k-nearest neighbors	1	5	15	30	59
Random assignment	923	4643	13969	27933	55190
Minimax declustering	280	1848	6434	13586	28832
Improvement (%)	70	60	54	51	48

Table 1. The number of k-nearest neighbor blocks assigned to the same disk

the number of blocks assigned to a disk and $Block_i$ and $Block_{i+1}$ are a pair of adjacent blocks on the disk. A high probability path length indicates that data blocks are clustered well on the disk and hence will require a small number of disk seeks for retrieval. When the *short spanning path* algorithm was used to cluster data blocks on each disk, the average probability path length was 23.7; the corresponding number for random block assignment was 13.3.

5.2. Simulation-based evaluation of data retrieval

Based on the common query patterns identified by earth science researchers [7, 9, 10, 24], we generated synthetic queries that uniformly cover the land masses of the world. We divided the land masses into 12 disjoint regions, in total covering almost all of six continents (excluding Antarctica). We computed two metrics: *model transfer time* and *model seek time*.

DEFINITION 1 *The* **model transfer time** *of a query q is defined as $\max_{i=1}^{M}\{N_i(q)\}$, where M is the number of disks used and $N_i(q)$ is the number of data blocks fetched from disk i to answer the query q.*

DEFINITION 2 *Given a query q, the* **model seek time** *is defined to be the number of clusters in the answer set of q. A cluster of blocks is a group of data blocks that are contiguous on disk.* [2]

The *model transfer time* has been considered a plausible measure of the actual block transfer time [6, 16] under the assumptions that: (1) all disks can be accessed independently, and (2) the volume of data is too large for caching. The *model seek time* was originally proposed in [12] as a measure of the clustering properties of space-filling curves [12, 17].

As was pointed out in [12], small gaps between fetched blocks are likely to be immaterial. Therefore, we use the total distance to be traveled by the disk arm, as well as the model seek time, to evaluate the clustering scheme.

We used 3-dimensional queries (two spatial dimensions and the temporal dimension). The sizes of queries are governed by a selectivity factor ($0 < r < 1$). The selectivity factor r denotes the percentage of the total spatio-temporal volume that the query covers. For example, a query for a region of size $L_{Lat} \times L_{Long} \times L_{Time}$ had the size $L_{Lat}\sqrt[3]{r} \times L_{Long}\sqrt[3]{r} \times L_{Time}\sqrt[3]{r}$. To simulate the data retrieval for these queries, we used the Titan index described in Section 3.2 and computed the model block transfer time and model seek time for each of the queries – without actually retrieving the data blocks. We used three values of selectivity: 0.01, 0.1 and 0.2 (1, 10 and 20 percent).

For each land region and each selectivity, we generated 500 range queries and computed the average model transfer time and the average model seek time. Table 2 and Table 3 show the results for all the land regions. In Table 2, the first two columns for each selectivity show the average model transfer time for random block assignment and the *minimax* algorithm respectively; and the third column shows the improvement achieved by *minimax*. In Table 3, the first two columns for each selectivity show the average model seek time for random assignment and SSP. To isolate the effects of clustering, the same scheme (*minimax*) was used to assign blocks to each disk.

We also measured the average distance over which the disk arm needs to move for each query. The *disk arm travel distance* is modeled by *(highest_offset - lowest_offset)* among the blocks in the answer set of a given query. For all the experiments, we observed an 11 to 97 percent improvement in the disk arm travel distance for SSP clustering relative to random block assignment.

5.3. Performance on the SP-2

We measured the performance of Titan for loading the database as well as for processing queries. We measured the loading performance by measuring the wall-clock time for the entire loading operation as well as for major sub-operations. To evaluate the query-processing performance, we ran a set of sample queries. Each of these queries generates a 10-day composite image, using the sensor measurements from bands 1 and 2. For simplicity, these sample queries are specified as rectangular boxes that cover land masses of varying sizes – the United Kingdom and Ireland, Australia, Africa, North America, South America, and the complete globe. The resolution of the output images for

[2] Contiguous data blocks may be considered to have contiguous logical block numbers, assuming that logical block numbers represent the relative locations of physical data blocks.

Selectivity	1 percent			10 percent			20 percent		
Declustering	random	minimax	impr.(%)	random	minimax	impr.(%)	random	minimax	impr.(%)
Land region A	10.9	7.9	28	27.6	20.6	25	38.3	28.5	26
Land region B	10.4	7.6	27	23.4	18.7	20	31.6	25.8	19
Land region C	11.2	8.1	27	26.8	22.0	18	36.5	31.0	15
Land region D	8.6	6.1	29	19.1	14.7	23	25.4	20.1	21
Land region E	55.3	46.3	16	119.9	105.0	12	150.4	136.0	10
Land region F	65.5	56.3	14	145.1	129.1	11	182.8	168.3	8
Land region G	9.1	6.7	26	32.3	25.6	20	54.2	44.6	18
Land region H	19.8	16.1	19	46.7	39.4	16	62.0	53.4	14
Land region I	16.1	12.4	23	36.9	31.6	14	49.6	43.1	13
Land region J	7.6	5.5	27	24.1	19.2	20	37.0	31.1	16
Land region K	5.2	3.4	33	15.4	10.7	31	23.4	16.6	29
Land region L	3.2	2.1	32	6.0	4.5	26	8.0	6.1	24

Table 2. Model transfer time

Selectivity	1 percent			10 percent			20 percent		
Clustering	random	SSP	impr.(%)	random	SSP	impr.(%)	random	SSP	impr.(%)
Land Region A	7.8	7.5	4	19.9	18.2	9	27.4	24.0	12
Land Region B	7.6	7.3	3	18.4	16.8	9	25.1	21.7	13
Land Region C	8.1	7.7	4	21.3	18.7	12	29.5	24.5	17
Land Region D	6.0	5.8	4	14.3	13.1	9	19.8	17.2	13
Land Region E	43.7	35.5	19	93.5	51.7	45	117.1	49.8	57
Land Region F	52.4	44.2	16	110.7	64.2	42	136.4	61.9	55
Land Region G	6.6	6.4	3	24.5	22.0	10	41.5	35.5	15
Land Region H	15.9	13.8	13	38.0	24.6	35	50.5	26.9	47
Land Region I	12.2	10.7	12	30.5	20.1	34	41.1	23.3	43
Land Region J	5.5	5.3	4	18.8	17.3	8	30.1	26.3	12
Land Region K	3.4	3.3	3	10.6	9.4	12	16.4	13.7	17
Land Region L	2.1	2.0	5	4.5	3.9	13	6.1	5.1	17

Table 3. Model seek time

all the sample queries was $100/128$ degrees of longitude by $100/128$ degrees of latitude.

5.3.1 Loading performance

Loading the database requires: (1) segmenting raw AVHRR files into data blocks; (2) building an index for the data blocks; (3) determining the data placement; and (4) copying the blocks to the appropriate locations. The first three phases were performed on a single node of the SP-2 and the last phase was done in parallel. For the 24 GB AVHRR data set, the entire loading operation took about four hours on the SP-2. The majority of the time was spent computing the data placement, which took about three hours to process the over 55,000 data blocks. The final step that moved data from the AVHRR files to the locations specified by the data placement algorithms took about twenty minutes.

5.3.2 Query-processing performance

We used the sample queries described above to evaluate the end-to-end performance of Titan and to understand the interactions between its subsystems. To study the interactions, we selectively disabled one or more of I/O, interprocessor communication, and computation. We studied four configurations. The first configuration (referred to as `I/O-only`) enabled only the block-read and block-read-check phases (without communication) and was intended to measure the disk I/O performance. The second configuration (referred to as `I/O+computation`) enabled the block-read, block-read-check and block-consume phases and was intended to estimate the extent to which use of asynchronous I/O was able to hide I/O latency. The third configuration (referred to as `I/O+comm`) enabled the block-read, block-read-check, block-send-check and block-receive phases and was intended to estimate the overlap the extent of overlap between I/O and communication. The fourth configuration (referred to as `full`) enabled all phases and was intended to measure end-to-end performance. The first two configurations were run with a single back-end node and did not generate the complete output image.

Figure 3 shows the results for the global query using the first two configurations. For the second configuration,

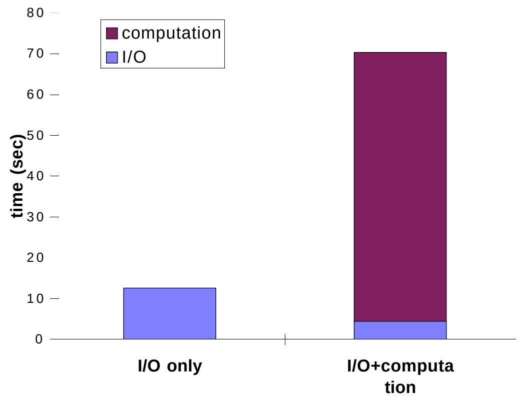

Figure 3. Resolving the global query using a single back-end node.

the time spent in performing computation is measured separately. To resolve the global query, the node retrieved 619 data blocks (116.2 MB) at an effective disk bandwidth of 9.2 MB/sec. Comparing this value to the maximum application-level disk bandwidth of 10.6 MB/sec achievable on an SP-2 node, as measured by a micro-benchmark, we note that the I/O overhead was small and that the data placement on a single node worked fairly well. The difference between the height of the left bar and that of the I/O part of the right bar indicates the amount of overlap between I/O and computation. In this experiment, the overlap was about 65% (8 sec).

We next focus on end-to-end performance and on the interaction between all three components. Figure 4 shows the execution times for resolving the sample queries for the third and fourth configurations. In each graph, the second and the third bar correspond to the performance for the third and the fourth configuration, respectively. For comparison, the leftmost bar in every graph shows the estimated disk-read time assuming a disk bandwidth of 10 MB/s. Table 4 shows the total amount of data read from disk and communicated between processors to resolve the sample queries.

By comparing the heights of the I/O parts of the two leftmost bars for each query, we note that when the query is large enough, a significant part of the disk-read time is overlapped with communication. When the query is small, however, each node only reads a few data blocks from its disks, so cannot achieve maximum disk bandwidth. That is why the estimated I/O times (at 10 MB/sec) for the Australia and United Kingdom queries are less than the measured times for overlapped I/O.

The rightmost bars in Figure 4 also show that computation did not overlap well with the combination of I/O and communication. We already know that I/O did overlap fairly well with computation. Therefore, we conclude that use of asynchronous communication did not hide communication latency. This is not surprising since the cost of communication in the MPL communication library available on the SP-2 is dominated by memory-copy costs especially on the so-called **thin** nodes on our machine. The memory copy also uses the processor, leaving no time for the processor to perform other computation during communication [25]. Snir *et al.* [25] did, however, report that slightly better overlap between computation and communication can be achieved with wide SP-2 nodes, which have higher memory bandwidth.

6. Discussion

The experimental results presented in Section 5 show that Titan delivers good performance for both small and large queries. Titan achieves interactive response times (less than 10 sec.) for local queries and relatively quick turnaround (1.5 min.) for global queries. Nevertheless, there are two factors that limit further improvements in performance. First, the high I/O parallelism has been achieved at the cost of locality. In the experiments, this shows up as a large amount of time spent in communication. Second, there is considerable computational load imbalance. Table 5 shows the minimum and maximum numbers of data blocks read and processed by different nodes for each of the sample queries.

The loss of locality is primarily due to the hidden assumption in the declustering algorithm, as well as in the algorithm that partitions the query processing, that the cost of moving the data from a disk to a processing node is the same for all disks and all nodes. In reality, data blocks retrieved by a back-end node must be forwarded to all consumers of the data block, resulting in a large amount of communication. As can be seen in Table 4, data blocks retrieved for the global query had an average of 1.4 remote consumers; the corresponding numbers for the Africa and the United Kingdom queries are 1.8 and 6.2 respectively. The computational imbalance is primarily due to the use of a uniform partitioning scheme to process AVHRR data that is distributed non-uniformly over the attribute-space. The non-uniform distribution is partly due to the structure of the satellite's orbit and partly due to the shape of the earth.

We are currently working on the trade-off between I/O parallelism and locality. We are considering two techniques: (1) using a two-phase strategy for composition of data blocks and (2) significantly increasing the size of the data blocks. The two-phase strategy would perform the processing and composition of all data blocks on a back-end node and forward only the composited result for combination with data from other nodes. This will reduce communication requirements. Since most of the processing for a data block will be done at the node on which it resides and since our decluster-

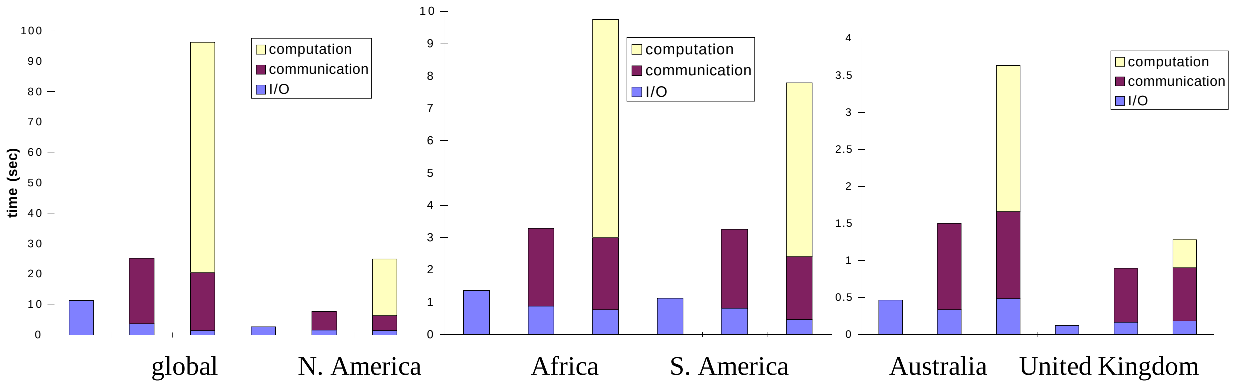

Figure 4. Resolving the sample queries with 15 back-end nodes.

sample query	total data read - # data blocks	total data communicated - # messages	ratio
global	9263 (1700 MB)	13138 (2500 MB)	1.4
Africa	1087 (203.3 MB)	2005 (374.9 MB)	1.8
North America	2147 (401.5 MB)	4313 (806.5 MB)	2.0
South America	896 (167.6 MB)	1740 (325.4 MB)	1.9
Australia	375 (69.9 MB)	997 (186.4 MB)	2.7
United Kingdom	97 (18.1 MB)	602 (112.6 MB)	6.2

Table 4. The total number of data blocks read and communicated to resolve the sample queries.

ing scheme achieves a good I/O balance, we expect this to significantly improve the computational balance. Increasing the size of the data blocks will increase locality and improve the performance of the local composition operations. It will, however, reduce parallelism, particularly for small queries.

7. Conclusions and Future Work

We have presented the design and evaluation of Titan, a high performance image database for efficiently accessing remotely sensed data. Titan partitions the data into coarse-grained chunks, and distributes the chunks across a disk farm. The system consists of a front-end, for query partitioning, and a back-end, for data retrieval and post-processing. The experimental results show that Titan provides good performance for queries of widely varying sizes.

We are currently investigating techniques for efficiently handling multiple concurrent queries. The issues that must be addressed include resource management and data reuse. Resource management issues arise from trying to optimize use of the limited amount of buffering space available on the processing nodes. Data reuse refers to scheduling processing of queries that overlap in space and/or time to achieve good system throughput.

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] Web site for AVHRR data from the NASA Goddard distributed active archive center. *http://daac.gsfc.nasa.gov/-CAMPAIGN_DOCS/FTP_SITE/readmes/pal.html*.

[3] L. T. Chen and D. Rotem. Declustering objects for visualization. In *Proceedings of the 19th VLDB Conference*, pages 85–96, 1993.

[4] D. DeWitt, N. Kabra, J. Luo, J. Patel, and J.-B. Yu. Client-server Paradise. In *Proceedings of the* 20^{th} *VLDB Conference*, 1994.

[5] K. Doan, C. Plaisant, and B. Shneiderman. Query previews in networked information systems. Technical Report CS-TR-3524, University of Maryland, Oct 1995.

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

[7] J. Eidenshink and J. Fenno. Source code for LAS, ADAPS and XID, 1995. Eros Data Center, Sioux Falls.

[8] M. T. Fang, R. C. T. Lee, and C. C. Chang. The idea of de-clustering and its applications. In *Proceedings of the 12th VLDB Conference*, pages 181–8, 1986.

query	data blocks read		data blocks processed	
	min	max	min	max
global	599	632	634	1095
Africa	65	79	117	151
North America	137	151	172	430
South America	50	66	89	148
Australia	19	28	56	80
United Kingdom	5	9	35	48

Table 5. The number of data blocks read and processed by the back-end nodes.

[9] S. Goward, B. Markham, D. Dye, W. Dulaney, and J. Yang. Normalized difference vegetation index measurements from the Advanced Very High Resolution Radiometer. *Remote Sensing of Environment*, 35:257–77, 1991.

[10] NSF/ARPA Grand Challenge Project at the University of Maryland for Land Cover Dynamics, 1995. *http://www.umiacs.umd.edu:80/research/GC/*.

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

[12] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of the 1990 ACM-SIGMOD Conference*, pages 332–42, May 1990.

[13] I. Kamel and C. Faloutsos. Parallel R-trees. In *Proceedings of the 1992 ACM-SIGMOD Conference*, pages 195–204, June 1992.

[14] S. Liang, S. Goward, J. Ranson, R. Dubayah, and S. Kalluri. Retrieval of atmospheric water vapor and land surface temperature from AVHRR thermal imagery. In *Proceedings of the 1995 International Geoscience and Remote Sensing Symposium, Quantitative Remote Sensing for Science and Applications.*, pages 1959–61, July 1995.

[15] D.-R. Liu and S. Shekhar. A similarity graph-based approach to declustering problems and its application towards parallelizing grid files. In *the 11th Inter. Conference on Data Engineering*, pages 373–81, Mar. 1995.

[16] B. Moon, A. Acharya, and J. Saltz. Study of scalable declustering algorithms for parallel grid files. In *Proceedings of the Tenth International Parallel Processing Symposium*, pages 434–40, Apr. 1996. An extended version is available as University of Maryland Technical Report CS-TR-3589.

[17] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of Hilbert space-filling curve. Technical Report CS-TR-3611, University of Maryland, Mar. 1996.

[18] B. Moon and J. H. Saltz. Scalability analysis of declustering methods for Cartesian product files. Technical Report CS-TR-3590, University of Maryland, Apr. 1996.

[19] S. Morehouse. The ARC/INFO geographic information system. *Computers and Geosciences: An International Journal*, 18(4):435–41, August 1992.

[20] J. Nievergelt and H. Hinterberger. The Grid File: An adaptive, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, Mar. 1984.

[21] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 181–90, Apr. 1984.

[22] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(11):1389–1401, Nov. 1957.

[23] C. Shock, C. Chang, L. Davis, S. Goward, J. Saltz, and A. Sussman. A high performance image database system for remote sensing. In *24th AIPR Workshop on Tools and Techniques for Modeling and Simulation*, Washington, D.C., Oct. 1995.

[24] P. Smith and B.-B. Ding. Source code for the AVHRR Pathfinder system, 1995. Main program of the AVHRR Land Pathfinder effort (NASA Goddard).

[25] M. Snir, P. Hochschild, D. Frye, and K. Gildea. The communication software and parallel environment of the IBM SP2. *IBM Systems Journal*, 34(2):205–221, 1995.

[26] M. Ubell. The Montage extensible datablade architecture. In *Proceedings of the 1994 ACM-SIGMOD Conference*, page 482, May 1994.

[27] T. Vijlbrief and P. van Oosterom. The GEO++ system: An extensible GIS. In *Proceedings of the Fifth International Symposium on Spatial Data Handling*, pages 40–50, August 1992.