Client–Server Paradise*

David J. DeWitt, Navin Kabra, Jun Luo, Jignesh M. Patel, and Jie-Bing Yu
Computer Sciences Department,
University of Wisconsin, Madison
paradise@cs.wisc.edu

Abstract

This paper describes the design and implementation of Paradise, a database system designed for handling GIS type of applications. The current version of Paradise, uses a client–server architecture and provides an extended–relational data model for modeling GIS applications. Paradise supports an extended version of SQL and provides a graphical user interface for querying and browsing the database. We also describe the results of benchmarking Paradise using the Sequoia 2000 storage benchmark.

1 Introduction

Over the last five years interest in Geographic Information Systems (GIS) has increased significantly. Existing systems represent an integration of ideas from many different fields including remote sensing, photogrammetry, and computer cartography [MGR91]. In turn, new application domains have placed additional demands on existing systems. For example, GIS systems are now being used to store and process vast amounts of remote–sensed data gathered from sensors on satellites. These satellites scan the surface of the earth, measuring certain electromagnetic properties of the surface. This information is then radioed down to a receiving station on the surface. Since this process is completely automated and continues non-stop, the data volumes are enormous. In addition to storing the raw data, the receiving station will also generally reprocesses the new data into a set of standard data products that are then made available to scientists around the world.

*This work was partially supported by NASA Contracts #USRA–5555-17 and #NAGW–3895 and by an IBM Research Initiation Grant.

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

In addition to having to store and manage large volumes of data, GIS systems must also be capable of handling a variety of different data and query types. For example, many GIS applications provide support for at least two forms of spatial data: raster and vector data. Raster data is usually represented as a two (or more) dimensional array of integer or floating point values, corresponding, for example, to readings taken by a sensor on a satellite. Vector data, on the other hand, is generally composed of a set of lines, representing, for example, the outline of a region. The type of queries posed in such a system will frequently include predicates involving spatial relationships, such as spatial overlap or containment. In addition to providing an expressive data model and query language, a GIS must also provide an efficient mechanism for performing operations on spatial data if it is to successfully process spatial queries on large volumes of data.

Existing GIS systems employ a variety of different architectures [MP94]. Some systems (e.g. GRASS [Sea92]) store all data in normal operating system files, providing a library of functions for retrieving, manipulating, and displaying data. From a traditional database perspective these systems are very limited in terms of functionality especially with respect to query optimization and processing, transaction support, concurrency control, and physical data independence. Other GIS systems [Mor92, HHK+93], employ a hybrid approach in which a traditional relational database manager is used to store non-spatial data with the spatial data going into either the file system (ARC/INFO) or a spatial data manager (Papyrus). While the hybrid approach has been quite successful, it complicates query optimization and execution, especially in a multiuser environment. The third approach, as exemplified by Postgres [SR86], Gral [Güt89] Montage [Ube94], GEO [VvO92], and Paradise uses a integrated approach in which all data is stored in the database system.

We began the Paradise (**Para**llel **D**ata **I**nformation **S**ystem) project in early 1993 [DLPY93] as a response to the challenges posed by the Sequoia benchmark [SFGM93]. The goal of the Paradise project is to apply the object–oriented and parallel database technology developed as part of the EXODUS [CDF+86] and Gamma [DGS+90] projects to the task of implementing a parallel GIS system capable of managing extremely large (multi-terabyte) data sets such as those that will be produced by the upcoming NASA EOSDIS project [Car92]. The project is focusing its resources on algorithms, processing, and storage techniques, and not on making new contributions to the data modeling, query language, or user interface domains. Paradise supports storing, browsing, and querying of geographic data sets. Its data model is an extended–relational data model, extended with raster, polygon, and polyline ADTs and typed references. An extension of SQL is provided to support ad–hoc queries over extents of persistent objects. Paradise uses SHORE [CDF+94] as its storage manager for persistent objects, and a graphical user interface that is built using Tk, a public domain X11 toolkit.

At the outset, we organized the Paradise project as two phases. The goal of first phase was to produce a client-server version of Paradise. The second phase of the project is to parallelize the Paradise server to operate on a shared–nothing [Sto86] multiprocessor (our target multiprocessor platform is a 64 processor/64 disk Intel Paragon). The first phase is now complete and is described in this paper. In addition to allowing us to "get our feet wet" in the GIS domain, phase one of the project has produced a usable, client–server version of the system whose performance and functionality is comparable to other integrated systems.

The remainder of the paper is organized as follows. Section 2 describes Paradise's data model and query language. The software architecture of the system, including several novel techniques for dealing with spatial data, is presented in Section 3. Section 4 contains a performance evaluation of the system using the Sequoia benchmark. Finally section 5 contains our conclusions and some future plans.

2 Data Model And Query Language

2.1 Data Model

Paradise provides an extended–relational data model for modeling GIS applications. Three type constructors are provided: extent, tuple, and reference. An **extent** consists of a set of objects of the same type. A Paradise database consists of one or more such extents. Objects themselves are defined using the **tuple** type constructor. Each attribute can be an instance of either a standard base type (i.e. integer, float, string, ...), one of the predefined GIS–specific abstract data types (ADTs) including polygon, polyline, and raster, or a typed **reference** to another object. Since extents themselves are typed objects, the use of references allows the definition of a fairly rich set of complex objects. In addition, one can define functions on the ADTs which can be used in the predicate of a query. The ADTs, their functions, and their operators (methods) are defined and coded in C++. The type system can be extended either through inheritance from existing ADTs or by defining new ones. New ADTs must first be registered with the catalog manager before being used to define new object types. As an example, consider the weather database shown in Figure 1. In the example, "Text", "Raster", "Date", "Polyline" and "Polygon" are some of the predefined ADTs.

While fairly rich, the Paradise data model is more restricted than what a full object-oriented database system would provide [ABD+89]. For example, Paradise does not directly support set–valued attributes. We made this simplifying decision in order to avoid many of the implementation complexities associated with a full object–oriented data model. When the SHORE implementation of the ODMG standard [Cat93] object–oriented data model ODL is operational, we plan on switching to it.

```
create extent Instrument(name String,
  type Integer, manual Text);

create extent CloudCover(
  cloudDensity Raster,
  measuringDevice ref Instrument, date Date);

create extent Rivers(name String,
  shape Polyline, flood_plain Polygon,
  water_level Integer, levee_status Integer);

create extent Cities(boundary Polygon,
    name String, population Integer);
```

Figure 1: Sample Paradise Schema.

2.2 The Query Language

As a query language, Paradise provides an extended version of SQL. To SQL we have added the ability to invoke methods defined on the ADTs, and the ability to follow inter-object references using the standard nested dot notation [Zan83] for accessing components of complex objects (i.e. x.y.z).

Consider the schema shown in Figure 1. To locate cities that could be affected by floods, one might pose the query (Note that this query performs a spatial join between the Cities and Rivers extents):

```
Select * from Cities, Rivers where
Cities.boundary overlaps Rivers.flood_plain
and Rivers.levee_status = "Weak"
```

This query might be executed by first selecting all the `Rivers` tuples that satisfy the predicate on `levee_status`. Then, if an index exists on the `boundary` attribute of the `Cities` extent, a nested–loops index join might be performed using the `flood_plain` attribute of the selected `Rivers` tuples to filter out the "non–matching" `Cities` tuples.

As another example consider the query for finding the current cloud cover over all "large" cities, for which a "severe thunderstorm" warning needs to be issued:

```
Select name, cloudDensity.clip(boundary)
from Cities, CloudCover
where boundary.area() > 900 and
  date = "9/15/94"  and
  cloudDensity.clip(boundary).average()>10
```

Here `area` is a function that is defined on the polygon ADT, `average` is a function defined on the raster ADT, and `clip` is a function on the raster ADT that takes a polygon as its argument. The query selects all `Cities` that have an area greater than 900 sq miles and "joins" it with the `CloudCover` tuple that was scanned on "9/15/94". Further, only those "join" result tuples that have an average `cloudDensity` greater than 10 units are produced as result tuples. The result tuples have two attributes—one is the city name, and the other is the raster image corresponding to the cloud cover over the city. Note the use of the dot notation in the expression `cloudDensity.clip(boundary).average()`. This expression implies that the function `clip` should be applied to the `CloudCover.cloudDensity` attribute. `Cities.boundary` provides the argument for this `clip` function, which returns a value of type Raster. To this return value, the `average` function is applied. The return value of this function is then used for evaluating the predicate (`... > 10`).

3 Software Architecture

3.1 System Overview

Version 1.0 of Paradise employs a conventional client–server architecture as shown in Figure 2. The server includes a tuple manager, an extent manager, a catalog manager, a query optimizer, and a scheduler. The front–end provides a graphical user interface that supports querying, browsing, and updating of Paradise objects through either its graphical or textual interfaces. In either case, the front–end transforms a query into our extended SQL syntax and ships it to the Paradise server for execution. After executing the query, the server ships the result objects back to the client process through a Postgres–like portal mechanism [Gro93]. All communication between the front–end and server processes is in the form of remote procedure calls running over TCP/IP.

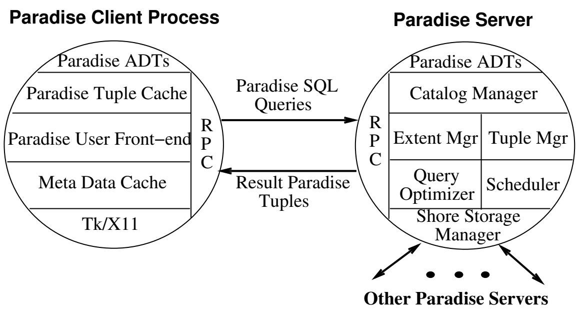

Figure 2: Paradise Process Architecture

3.2 Paradise User Front-end

Although the user interface is an important component of any database system, it is an especially important part of a GIS. In particular, a GIS must provide a convenient graphical interface for the visualization and manipulation of spatial data. The front–end should be capable of graphically querying, browsing and updating spatial objects stored in the database. It should also be able to address the various complex user requirements for spatial processing and analysis. In this section, we will describe our approach to developing such an user front–end for Paradise.

Our first attempt at a front–end used GEO, a C++ based, graphical user interface for geographical database systems [vOV91]. GEO uses the ET++ class libraries [FW91] (based on X11) as its display vehicle and Postgres [SR86] as its underlying spatial database management system. GEO provides both a graphical browser for viewing spatial data and a graphical interface for composing ad–hoc queries.

We converted GEO to use Paradise instead of Postgres as its database server. While this approach enabled us to rapidly produce a working graphical user interface for Paradise, we encontered a number of significant problems. First, since GEO was designed specifically to run on top of Postgres, each object returned from the Paradise server to the GEO front–end had to be converted from its Paradise representation to the corresponding Postgres representation.[1] Needless to say, the performance of this approach was not very good. A more serious problem was that Version 1.33 of GEO cannot display multiple spatial attributes; for example, an object with both polygon and point attributes. Third, GEO requires that the result of a join contain attributes from only one of its input relations. Finally, the modified ET++ library that GEO uses as its display library is extremely complex, not in the public–domain, and not available for a wide variety of

[1] The obvious solution, converting GEO to understand the Paradise object format, was determined to be far too difficult.

platforms.

Given these limitations we reluctantly decided that the best solution was to write our own interface. The approach was simple: "clone" GEO's "look and feel" while avoiding the limitations of the current GEO implementation. The new Paradise front–end is implemented using Tk [Ous91], a publicly available X11 toolkit based on a lightweight interpretive command language Tcl [Ous90]. Using Tk, instead of ET++ or Interviews [LCV88], resulted in a dramatic reduction[2] in the size and complexity of the front–end, without apparently sacrificing performance. The key features of the Paradise front–end include:

- Display of objects with spatial attributes on a 2–D map. For objects with multiple spatial attributes, one of the spatial attributes can be used to specify the position of the object on the screen. The spatial ADTs currently supported include points, closed polygons, polylines, and raster images.
- Layered display of overlapping spatial attributes from different queries or extents. For example, one can display city objects that satisfy a certain predicate (e.g. population > 300K) in one layer on top of a second layer of country objects.
- Querying through a graphical interface: implicitly issuing spatial queries by zooming, clicking, or sketching a rubber–banded box on the 2–D map.
- Querying by explicitly composing ad-hoc queries in Paradise's extended SQL syntax.
- Browsing the objects from an extent. In this mode attributes are displayed as ASCII strings.
- Updating Paradise objects. The object(s) to be updated can be selected either by pointing–and–clicking on the 2–D map or by selecting via the textual browser.
- General catalog operations including browsing, creating new databases, defining new extents, creating indices on attributes, and bulk loading data into extents from the Unix file system.

The structure of the Paradise user front–end is shown in Figure 3. It consists of the following components:

- The **Map View** is responsible for displaying and manipulating objects contained in one or more layers. The current position of the cursor is continuously displayed in a sub–window in units of the map projection system. Users can point and click on displayed objects to view their non-spatial attributes. In addition, users can also zoom into a selected region by sketching a rubber–banded box.
- The **Layer Manager** is responsible for adding, deleting, hiding, and reordering layers displayed by the Map View. Each layer corresponds to an extent of objects produced by executing some query.
- The **Extent browser** allows a user to view any Paradise extent and adjust the way it should be displayed by the Map View. The selected extent becomes a new layer with its spatial attributes displayable via the Map View.
- The **Query composer** allows a user to compose a SQL query using a simple text editor.
- The **Query executor** is the interface to the Paradise server. It ships SQL queries to Paradise server for execution and retrieves result tuples into its own object cache.
- The **Object cache** caches the result of a query in formats understood by Tcl/Tk.
- The **Meta cache** stores the catalog information of the currently open database.

[2] Approximately 75% reduction in the number of lines of code

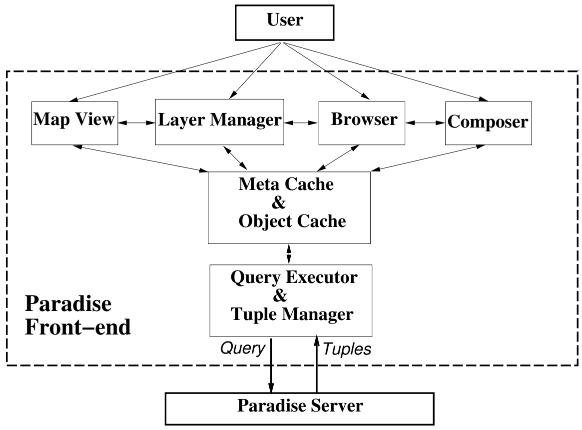

Figure 3: Architecture of the Paradise Front-end

A screen dump from the Paradise front–end is shown in Figure 4.

3.3 The Paradise Server

The Paradise server uses SHORE [CDF+94] as its underlying persistent object manager. The Paradise server is implemented as a SHORE Value Added Server (VAS) directly on top of the SHORE Storage Manager. To the basic SHORE server, Paradise adds a catalog manager, an extent manager, a tuple manager, a query optimizer and execution engine, and support for point, polyline, polygon, and raster ADTs. Since the SHORE server has been designed to run on shared–nothing multiprocessors, the task of extending

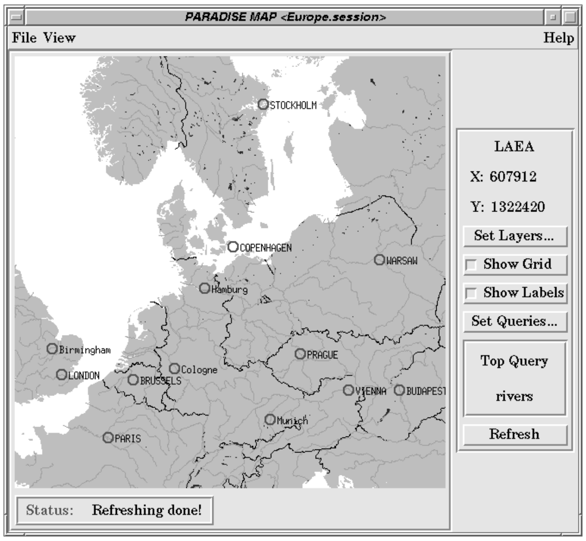

Figure 4: Map View of Paradise Front-end

Paradise to such an environment will be significantly simplified.

Execution of a query in Paradise proceeds as follows. After submission, the query is sent by the front–end to the Paradise server for execution. Here the query is parsed and optimized and an execution plan is generated. The Parser and Optimizer consult the Catalog Manager to obtain the necessary type information and statistics. Once a query plan has been generated, the plan is forwarded to the Query Scheduler and Executor for execution. As result tuples are produced, they are packed into pages and shipped to the client process for display and subsequent manipulation. Upon arrival at the client process, objects sometimes undergo further transformations (e.g. coordinate projection conversion) prior to display processing.

In designing and implementing the Paradise server, careful attention was paid to insure that the system could efficiently process queries (especially those involving spatial attributes) on large volumes of data. In the following sections, we describe several of the more interesting design and implementation issues that we encountered during the implementation phase.

3.3.1 Spatial Access Through R*–Trees

In order to support the efficient retrieval of objects with spatial attributes, R*–trees [BKSS90] (with full concurrency control and recovery) were added to the SHORE storage manager. R*–trees were selected because of their efficacy and "relative" ease of implementation, especially since we could reuse a lot of the existing SHORE B^+–tree code. Grid files [NHS84] and KDB–trees [Rob81] were not considered as these multidimensional access methods do not do a good job of handling non–point spatial data [Gre89]. R^+–trees [TS87] (another variant of the R–tree [Gut84]) reduce the overlap between nodes by duplicating spatial objects across different nodes. However, when a full node in an R^+–tree is split, the split must be propagated in both a downwards and upwards direction. This significantly complicates implementing concurrency control and recovery. Finally, R*–trees provide support for "forced reinsert" [BKSS90], which makes it possible to dynamically re–clustering spatial objects in the index. We feel that this feature is very important in order to avoid performance degradation in a dynamically changing environment.

3.3.2 Bulk Loading R*–Trees

An important characteristic of any access method is its ability to perform initial index construction via a bulk load. This is especially important in a system like Paradise that is designed to efficiently handle very large volumes of spatial data. With respect to R*–trees, this requires being able to both load data at a fast rate and to produce a good clustering of the rectangles in the resulting index. Many systems, instead of bulk loading the R–tree, use multiple insertions, one per tuple. This results in very long load times since the R–tree is split repeatedly during insertions. Bulk loading builds the index bottom up and guarantees that each index page is only processed once.

In order to improve the bulk load time while retaining the effectiveness of the resulting R*–tree, a tree packing algorithm must be used. Like [FK93], our bulk load algorithm does spatial sorting using the Hilbert Curve. The Hilbert Curve was selected as it has better performance than other spatial ordering curves (e.g. Z–ordering, Grey code, column–scan) in a spatial query processing domain [Jag90]. However, unlike [FK93], our algorithm does not pack the leaves of the R*–tree to 100% utilization as we discovered (through simulation) that doing so may not generate a well structured R*–tree when the input data is not distributed uniformly. Our algorithm uses two special mechanisms to make it more resilient to different spatial data distributions:

I) A Heuristic Approach to Rectangle Packing

To guide the packing process, a heuristic is used to decide when to stop adding entries to the current node and to move on to the next node. The heuristic uses two parameters, a fill factor for monitoring the utilization of the current node and an expansion factor for measuring the increase in size of the minimum bounding box for the node that would occur if the next rectangle were added. When the fill factor reaches a minimum threshold (e.g. 75%) and the expansion factor reaches a maximum threshold (e.g. 120%), the packing process flushes the current node and starts adding entries to the next node. This optimization is designed to achieve spatial clustering by packing spatially "close" objects together into the same node to the maximum extent possible, even if it means incurring some decrease in storage utilization. Consequently, spatially clustered nodes will have less overlap between their minimum bounding boxes.

II) Caching with Repacking

As each packed node is produced it is added to a small (e.g. size 3) cache of recently packed nodes that have not yet been written to disk. Since the nodes in the cache were packed independently of one another, their minimum bounding boxes may overlap with one another. The rectangles from each of the nodes in the cache are then inserted into a single large node. This node is then resplit into smaller nodes using the standard R*–tree splitting algorithm. This process of combining and then splitting improves the spatial clustering and minimizes the overlap between the nodes in the cache. Finally, the cached node with the smallest value on the Hilbert Curve is flushed to disk, leaving room for the next node to be produced.

3.3.3 Implementation of Paradise ADTs

Each Paradise ADT has three different representations: an *in–memory* format, a *database* format, and an *external* format. The *in–memory* representation is the format in which the ADTs are stored in user space; The *database* representation is the format in which they are stored on disk/tape in the database; and the *external* representation is an ASCII representation of the data used for either input (*e.g.* during initial loading) or output (*e.g.* as the output of a query being viewed through the query browser in the front–end). All ADTs have conversion methods to switch between the different representations. A base ADT type is used as the super class of all Paradise ADTs. This super class provides a set of low–level memory management routines for memory–resident ADT instances plus a standard interface for the common conversion modules.

Paradise ADTs can be classified into two broad categories: spatial and non–spatial. The spatial ADTs (points, polygons, and polylines) all provide spatial methods and operators to deal with spatial analysis such as overlap, containment, and adjacency. In addition, each of these operators can be applied to different types of spatial ADTs (e.g. to determine whether a polygon and polyline overlap). Spatial functions such as "minimum bounding box" and "geometric size" calculations are also supported. Each spatial ADT instance also stores its coordinate projection system and the ADT classes provide methods for converting between different projection systems. The raster ADT provides several unique operations including polygon clip and "lower_resolution". The raster ADT in described in more detail below.

Raster ADT

Raster images tessellate space into regular shaped cells and assign a value to each cell. The value of each cell generally corresponds to the readings of some satellite sensor. Raster images, specially those used for studying large portions of the earth surface, can thus be very big. For example, the National Oceanographic and Atmospheric Administration (NOAA) Advanced Very High Resolution Radiometer (AVHRR) has a cell size of approximately 1.1km x 1.1km (at the nadir) [MGR91]. If the size of each cell value is 2 bytes, each raster image for a region corresponding to the United States (5500km x 3000km) will consume about 27 MBytes of space. In order to make operations on such large images as efficient as possible, the raster ADT in Paradise employs several techniques to improve performance. These techniques are described in the following two sections.

I) Separation of Raster Header and Data

When implementing the raster ADT in Paradise, we decided to break each raster ADT instance into two pieces: a *raster header* and the actual *raster image*. The raster header is used to store descriptive data about the raster image. The actual raster image, which consists of a two dimensional array of values (one per cell) is stored as a separate object in the SHORE storage manager.

The raster header contains the SHORE OID (object identifier) of the corresponding raster image, the size of the raster image, and the bounding box of the raster image.

As an example, consider Figure 5 which shows how Paradise stores the objects of the `CloudCover` extent of the weather database (see Figure 1 for the database schema). Each `CloudCover` instance has three attributes: `date` of type Date, `measuringDevice` which is a reference to an Instrument object, and `cloudDensity` which is of type Raster. Physically each object in the extent consists of three values: a date, the OID of the instrument used to take this measurement, and the raster header for the cloudDensity attribute. The objects containing the raster images for the CloudCover extent are themselves stored as large objects in a separate SHORE file.

This approach has a number of significant advantages. First, as illustrated by Figure 5, the objects

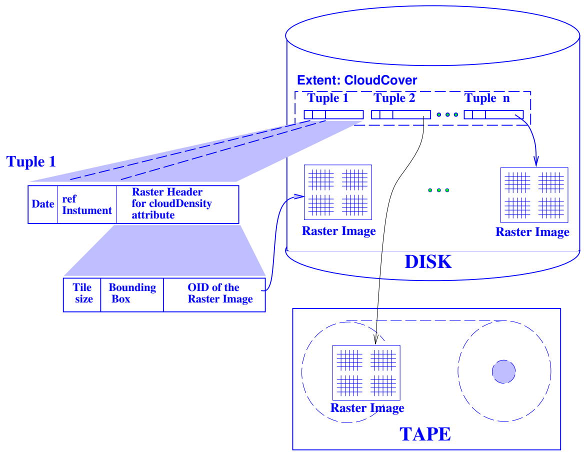

Figure 5: Physical Representation of the CloudCover Extent

containing the actual raster images can transparently migrate between secondary and tertiary storage. Second, by storing the raster images as large objects in a separate SHORE file, the tuples in the primary extent remain physically clustered with one another, significantly improving the performance of a sequential scan over the extent. Finally, even for queries involving the raster attribute, the raster images need not always be brought into memory. For example, consider a clip operation between a polygon and a raster attribute. To determine, whether an object satisfies the predicate we only need to check the bounding box information stored in the raster header part of the tuple. Even if the tuple does satisfy the clip predicate the raster images are fetched "lazily" – only when the image actually needs to be manipulated or displayed.

To further enhance performance of operations on raster images, each raster image is actually decomposed into regular rectangular shaped regions called *tiles*. The data in each tile is stored as a separate SHORE object. A map table (one per each raster image) is used for maintaining the correspondence between the tile objects and the region of the raster image corresponding to that tile object. The raster header simply stores the OID of the map table object.

Decomposition of the raster image into tiles allows Paradise to fetch only those portions that are required to execute an operation. For example, consider Figure 6, which illustrates the raster image being clipped by a polygon (as required by query 2 in Section 2.2). When the raster attribute is first needed for performing the clip operation, only the mapping information for the raster image is read from the disk. The OID part of the raster header is "swizzled" to point to the in–memory mapping information. From the spatial position of the polygon and the mapping table, we can precisely calculate the tiles of the raster image that

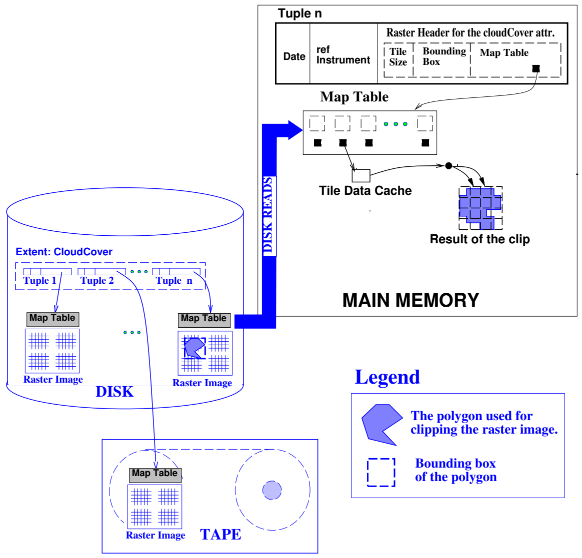

Figure 6: Processing the Clip Function for Attributes of Type Raster

are needed by the clip operation. Then, each relevant tile of the input is read from the disk and processed by the clip operation.

II) Compression as an Optimization Strategy

While compression techniques have been widely used in many image processing domains, only occasionally have they been integrated directly into a database system [SWKH94, GS91]. Several problems arise when such an integration is attempted. First, the unit of compression is generally the entire image. This approach makes sense if the entire image is always required. However, if only a piece of the image is needed, the cost of uncompressing the entire image may overshadow the improvement in performance resulting from having to read less data from disk. The situation is even worse if part of the image is updated as the entire object will have to be read, uncompressed, updated, recompressed, and written back to disk. A second problem is the unpredictability of the effectiveness of the compression algorithms in handling various kinds of data. If the compression ratio (defined as $\frac{data\ size\ without\ compression}{data\ size\ with\ compression}$) is too low, then the added cost of compression/decompression process could degrade overall system performance.

In Paradise, we combine lossless compression techniques with decomposition to solve the first problem. As discussed above, raster images are stored on secondary (and tertiary) storage as a number of smaller tiles. Each tile serves as the basic unit of compression. The raster image now becomes a large object with compressed tiles as its subcomponents. The mapping between a particular position in the image and its associated tile is performed via the mapping table (which is stored along with the tiles). To handle the unpredictability of the compression algorithm, we monitor the effectiveness of each compressed tile as it is created. If compression does not reduce the size of the tile significantly, we store the tile in its uncompressed form (a flag in the mapping table is used to indicate whether or not a tile is compressed).

Currently, only the basic LZW algorithm [Wel84] is used for compression and all raster objects are decomposed into rectangular shaped tiles. In the future, we plan on adding fancier, domain specific compression algorithms. We are also considering adopting the Quadtree [Sam89] approach, which has the additional advantage of improving the performance for certain types of spatial analysis on raster objects.

4 Performance Evaluation

To evaluate the performance of Paradise, we used the Sequoia 2000 Storage Benchmark [SFGM93]. The Sequoia benchmark uses real data sets and defines a suite of 11 queries that were chosen to be representative of the queries that earth scientists frequently pose to such a system. The benchmark has four different scales of data. For the purpose of benchmarking Paradise, we chose the **regional** benchmark. The data for this benchmark is fairly big (just over 1GByte) and can fit on a single disk. At later stages in the project, we intend to run the **national** and, perhaps, the **earth** benchmark. The **national** benchmark is around 18 GByte and the **earth** benchmark is multiple terabytes. While the national benchmark will fit on a moderate–size secondary storage system, the earth benchmark clearly requires the use of a tertiary storage system.

A brief description of the regional benchmark follows (for more details, readers are referred to the original benchmark paper [SFGM93]).

4.1 Description of the Regional Benchmark

The regional benchmark comprises of data corresponding to a 1280km X 800km rectangular region, covering parts of California and Nevada. The data set for this benchmark primarily consists of the following different data sets.

- **Raster data.** This corresponds to the readings of the earth surface taken by sensors on a satellite. The raster image consists of a 16 bit value for each cell of the area being scanned. The size of each cell is 0.5km X 0.5km and hence each raster image is about 8MBytes. Each image has a time field (when the reading was taken) and a frequency field (frequency of the instrument taking the reading) associated with it. The raster data set contains a total of 130 such readings.
- **Polygon data.** This consists of a set of regions, the boundaries of which are defined using a collection of lines. Each region has an integer typed landuse value associated with it.
- **Point data.** This consists of (location, name) pairs, which correspond to geographic points that have specific geographic features.
- **Directed Graph data.** This data set contains information about drainage networks. Each river is represented as a collection of line segments.

The Paradise schema for the Sequoia benchmark consists of the following extents

```
create extent raster (time Integer,
  frequency Integer, data Raster);
create extent polygon (landuse Integer,
  shape ClosedPolygon);
create extent point (location Point,
  name String);
create extent graph (shape PolyLine);
```

A brief description of the queries $1 \ldots 10$ follows (for more details see [SFGM93]). Terms in a query in all capitals (e.g. FREQ, RECT ...) are constants.

Query 1: Loads all the data files and builds a clustered R^*–tree on "point.location" and "polygon.shape". Non–clustered B^+–tree indices are constructed on "raster.frequency", "raster.time", "point.name" and "polygon.landuse".

Query 2: This involves clipping a portion of the raster images taken by a certain sensor.

```
select raster.data.clip(RECT), raster.time
from   raster where  frequency = FREQ
```

Query 3: This computes the average of the clipped portion of the raster images taken at a certain time.

```
select average (raster.data.clip(RECT))
from   raster  where  time = TIME
```

Query 4: This query selects one raster image (there is only one raster image for a given time and frequency), which is then clipped to the rectangular region under study. The result tuple is stored at a lower resolution.

```
create extent rasterTemp (time Integer,
  frequency Integer, data Raster);

insert into rasterTemp
select time, frequency,
       data.clip(RECT_VAL).lower_res(RES_VAL)
```

```
from raster
where time = TIME and frequency = FREQ
```

Query 5: This query selects a given point.

```
select * from  point where name = POINT-NAME
```

Query 6: This query select and stores all polygons that overlap with the specified rectangular region.

```
create extent polygonTemp (landuse Integer,
  shape ClosedPolygon);

insert into polygonTemp
select * from polygon
where shape overlaps RECT
```

Query 7: This query finds all polygons, greater than a certain area, that are contained in a circle.

```
select * from  polygon
where shape containedIn Circle(LOC, RADIUS)
 and  shape.area() >  AREA;
```

Query 8: This query selects all polygons that overlap a rectangular region around a point. Note this query involves a spatial join between the point and the polygon data.

```
select polygon.location, polygon.landuse
from   polygon, point
where  point.name = POINT-NAME
 and   polygon.shape overlaps
          point.location.makebox(SIDE_VAL)
```

Query 9: This query selects all raster images corresponding to polygons with a certain landuse. This query is a spatial join between the polygon and the raster extents.

```
select polygon.shape,
       raster.data.clip(polygon.shape)
from polygon, raster
where polygon.landuse  = LANDUSE
 and  raster.frequency = FREQ
 and  raster.time      = TIME
```

Query 10: This query again performs a spatial join between the point and the polygon data, selecting all points that overlap polygons with a specified landuse value. The query is executed as two parts. In the first part we select all the points that overlap with the selected polygons. In the next part we remove from the selected points those points that overlap with any islands.

```
create extent pointsFoo(location Point,
  name String);
create extent pointsResult(location Point,
  name String);

insert into pointsFoo
select distinct point.name, point.location
from   polygon, point
where  polygon.landuse = LANDUSE and
  polygon.shape overlaps point.location

insert into pointsResult
select * from pointsFoo minus
select distinct pointsFoo.name,
  pointsFoo.location
from   islands, pointsFoo
where  islands.shape overlaps
              pointsFoo.location
```

4.2 Effectiveness of Compression in Conjunction With Tiling

In this section, we evaluate the effectiveness of compression and the choice of a tile size for the raster ADT. The use of compression has the benefit of reducing the amount of data that is stored and read from the disk. On the other hand, using compression incurs a CPU overhead for compressing and decompressing the data. Furthermore, compression is more effective (yields larger compression ratios) if the unit of compression is large. Since we use a tile as the basic unit for compression, this argues for larger tile sizes. However, using a larger tile size implies that operations like the clip operation will fetch more redundant data.

To quantify these tradeoffs, we ran the raster queries (queries 2, 3, 4 and 9) of the Sequoia 2000 benchmark for three configurations. The first configuration used a tile size of 8KB, the same as SHORE's page size. This configuration represents the best case for not using compression. The relatively small tile size minimizes the amount of redundant data that is read from disk. The second configuration used a very large tile size of 512KB in conjunction with compression. The last configuration, which lies somewhere in the middle of the spectrum, used compression with a tile size of 128KB.

The execution time for running queries 1, 2, 3, 4 and 9 are shown in Table 1. (The system configuration used here was the same as that described in Section 4.4.)

As can be seen from Table 1, using compression increases the database loading time (query 1). For the 512KB tile size, the average compression ratio was observed to be about 1.85, implying a 46% reduction in the amount of data that was written to the disk. The 128KB tile size configuration had a similar compression ratio (of about 1.75, implying a 43% reduction in the amount of data that was written to the disk).

Query #	No Compr. 8KB tiles	Compr. 128KB tiles	Compr. 512KB tiles
1	3019.9 sec	3613.0 sec	4104.7 sec
2	18.0 sec	13.1 sec	18.3 sec
3	2.3 sec	2.0 sec	3.0 sec
4	0.6 sec	0.6 sec	0.7 sec
9	2.8 sec	2.8 sec	4.7 sec

Table 1: Effect of Compression and Tile Size

While compression reduced the amount of data that got written to the disk, the CPU overhead for compression outweighed the savings in disk I/O thereby increasing the overall load time.

Looking at the query execution times in Table 1, we observe that as we move from a 8KB tile size to the configuration using a 128KB tile size with compression, the query execution times almost always improve. However, moving to a 512KB tile size degrades the performance because more redundant data gets fetched, while the compression ratio increases only slightly.

4.3 Effectiveness of Building Clustered Spatial Indices

As mentioned in Section 3.3.2, Paradise provides a mechanism for building spatially–clustered R^*–trees. Clustered indices have the advantage that fewer data pages need to be fetched (as objects "close" to each other are on the same page). On the other hand, building a clustered spatial index requires that the tuples be clustered based on their spatial position. A non–clustered index, however, only requires sorting the (oid, bounding box) pairs for each tuple. Thus a clustered spatial index speeds the evaluation of queries, while incurring a load time penalty. To quantify the tradeoffs, we ran an experiment that had the following two parts: one in which a clustered index was created on the `shape` attribute of the `polygon` extent, and the other in which a non–clustered index was created on the same attribute. The results of this experiment are shown in Table 2.

Query #	Non-Clust. R*–tree	Clust. R*–tree	Speedup
Load	318.2 sec	559.0 sec	– 43.1 %
6	8.2 sec	7.4 sec	10.8 %
7	0.2 sec	0.2 sec	0.0 %
8	8.7 sec	7.2 sec	20.8 %

Table 2: Effect of clustering

The load time shown above includes the time to load the polygon data and the time to build the R*-tree. We observe that for a 43% penalty in loading the data (which has to be done only once), we obtain a 21% performance improvement for Query 8. Query 7 retrieves only one polygon, and as a result there is no difference between the two cases. All of these queries have a low selectivity (less than 1%) and hence retrieve very few tuples. With a larger selectivity, we would have observed a bigger difference in the performance of the two cases.

4.4 Comparison with Other Systems

In this section, we compare Paradise with two other systems, namely POSTGRES [SR86] and Illustra (formerly called Montage [Ube94]). In [Sto94] it was shown that these systems outperformed GRASS and IPW, two popular GIS systems. The machine used for the benchmark was a Sun SPARC–10/40 with 32 MBytes of memory, running SunOS Release 4.1.3. One Seagate 2GByte disk (3.5" SCSI, model # ST 12400N) was used to hold the database. (Both Illustra and POSTGRES used a UNIX file, while Paradise used a raw disk for its data volume. Neither Illustra nor POSTGRES supports the use of a raw disk for the data volume). A second Seagate 2GByte disk was used to hold the raw input data and the log for each system. The binaries of each system were stored on a third 1GB disk (3.5" SCSI, model # ST 11200N), which also served as the system swap disk. About 200MB of free disk space was left on this disk to ensure that none of the systems paid an unrealistically high cost due to swapping.

We used version 4.2 of POSTGRES and version 1.3 of Illustra. For Paradise, the use of compression was turned on and a tile size of 128KB was used for the raster ADT. Both Paradise and Illustra were run with a 16 MB buffer pool. The performance of POSTGRES was better with 0.5 MB buffer pool than with a 16MB buffer pool. Hence, a 0.5MB buffer pool was used for POSTGRES.

The polygon data was pre–processed, so that very large polygons (> 500 points) were broken up into smaller polygons. This was done because the current version of Paradise[3] and the version of POSTGRES that we were using could not handle these very large polygons. Although Illustra could handle them, to ensure a fair comparison, we used the same pre–processed data for all the systems.

For all the systems, we ran all the benchmark queries five times and took the average of the middle three numbers. Each run was taken by starting up a client that sequentially issued all the queries in the benchmark. Further, each query was run as a separate transaction. Thus for the five runs, we ran the client

[3]Paradise can handle large objects, but currently has some problems with the spatial sorting of large objects.

executable five times in a row.

All the scripts that were used for running the benchmark are available via anonymous ftp from the Paradise directory of `ftp.cs.wisc.edu`. The scripts for POSTGRES and Illustra are modified versions of the scripts that we had received from the developers.

A few modification were made to the scripts provided by POSTGRES and Illustra in order to exactly match the the benchmark originally specified in [SFGM93]. For example, the scripts that we received used a value of 50 meters for the constant `SIDE_VAL` in query 8 (refer to section 4.1 for the query). However, the value specified for this in the original benchmark [SFGM93] is 50,000 meters.

Paradise cannot run query 10 because the minus operator is not currently implemented.

The numbers for the three systems are shown in Table 3.

Query #	Paradise	Illustra	POST–GRES
1	3613.0 sec	5748.0 sec	8687.0 sec
2	13.1 sec	14.6 sec	13.4 sec
3	2.0 sec	4.8 sec	5.4 sec
4	0.6 sec	2.4 sec	1.3 sec
5	0.2 sec	1.0 sec	0.9 sec
6	7.0 sec	20.5 sec	36.0 sec
7	0.6 sec	1.2 sec	30.5 sec
8	9.4 sec	23.7 sec	62.2 sec
9	2.8 sec	1.1 sec	2.8 sec
10	—	0.6 sec	327.2 sec

Table 3: Sequoia Benchmark numbers.

As can be seen, except for query 9, Paradise generally has the best performance. The raster queries (2, 3, and 4) benefit from the use of performance enhancing techniques like tiling and compression, while the polygon and the point queries (6, 7 and 8) benefit from the use of clustered indices on spatial attributes.

5 Conclusions and Future Directions

This paper describes the client-server version of Paradise, a new GIS under development at the University of Wisconsin. Paradise provides an extended-relational data model with support for point, raster, polygon, and polyline ADTs, and an extended version of SQL for formulating ad–hoc queries. A graphical user interface based on the Tk toolkit allows the user to query and browse graphically.

To facilitate handling large collections of large raster, satellite images, Paradise incorporates several performance optimizations including the transparent separation of raster images from their associated metadata, division of raster images into tiles to minimize unnecessary I/O, and the automatic application of lossless compression/decompression on a tile–by–tile basis. Paradise's performance is competitive with other systems when executing queries from the Sequoia benchmark.

During the next phase of the project we will add support for tertiary storage and extend the software to run on "shared nothing" multiprocessors [Sto86].

6 Acknowledgement

We would like to thank C. K. Tan and Mike Zwilling for patiently answering our questions regarding the SHORE storage manager, the developers of the Sequoia benchmark for providing us with a realistic benchmark with real data sets, Paul Brown for his help with Postgres and Shel Finkelstein for his assistance with Illustra. We would also like to acknowledge Joe Hellerstein for answering some of our Postgres questions and in helping us install Montage on our machines. Jeff Naughton deserves special thanks for his many useful suggestions during the course of this project.

References

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. "The Object-Oriented Database Manifesto". In *International Conference on DOOD*, Japan, 1989.

[BKSS90] N. Beckmann, Hans-Peter Kriegel, R. Schneider, and B. Seeger. "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles". In *Proceedings of the 1990 ACM-SIGMOD Conference*, June 1990.

[Car92] R. V. Carlone. "NASA's EOSDIS Development Approach". Technical report, United States General Accounting Office, February 1992.

[Cat93] R. G. G. Cattell, editor. *"The Object Database Standard: ODMG–93"*. Morgan Kaufmann Publishers, San Mateo, California, 1993. With contributions by T. Attwood, J. Duhl, G. Ferran, M. Loomis and D. Wade.

[CDF+86] M. J. Carey, D. J. Dewitt, D. Frank, G. Graefe, M. Muralikrishna, J. E. Richardson, and E. J. Shekita. "The Architecture of the EXODUS Extensible DBMS". In *Proceedings of the 12th VLDB Conf.*, September 1986.

[CDF+94] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. Tsatalos, S. White, and M. J. Zwilling. "Shoring up Persistent Objects". In *Proceedings of the 1994 ACM-SIGMOD Conference*, "Minneapolis, Minnesota", May 1994.

[DGS+90] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. "The Gamma Database Machine Project". *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[DLPY93] D. J. DeWitt, J. Luo, J. M. Patel, and J. Yu. "Paradise – A Parallel Geographic Information System". In *Proceedings of the ACM Workshop on Advances in Geographic Information Systems*, Arlington, Virginia, November 1993.

[FK93] C. Faloutsos and I. Kamel. "Packed R–Trees Using Fractals". In *Conference on Intelligence and Knowledge Management*, November 1993.

[FW91] E. Famma and A. Weinand. *"ET++3.0–Introduction and Installation"*. UBILAB, Union Bank of Switzerland, 1991.

[Gre89] D. Greene. "An Implementation and Performance Analysis of Spatial Data Access Methods". In *Proc. of the 5th Data Engineering Conf.*, 1989.

[Gro93] The Postgres Group. *"Postgres 4.1 Reference Manual"*. University of California, Berkeley, CA, 1993.

[GS91] G. Graefe and L. D. Shapiro. "Data Compression and Database Performance". In *Proceedings of the ACM/IEEE-Computer Science Symposium on Applied Computing*, 1991.

[Gut84] A. Gutman. "R-trees: A Dynamic Index Structure for Spatial Searching". In *Proceedings of the 1984 ACM-SIGMOD Conference*, Boston, Mass, June 1984.

[Güt89] R. H. Güting. "Gral: An Extensible Relational Database System for Geometric Applications". In *Proceedings of the 15th VLDB Conf.*, August 1989.

[HHK+93] W. Hasan, M. Heytens, C. Kolovson, M. A. Neimat, S. Potamianos, and D. Schneider. "Papyrus GIS Demonstration". In *Proceedings of the 1993 ACM-SIGMOD Conference*, Washington, D.C., May 1993.

[Jag90] H. V. Jagadish. "Linear Clustering of Objects with Multiple Attributes". In *Proceedings of the 1990 ACM-SIGMOD Conference*, May 1990.

[LCV88] M. A. Linton, P. R. Calder, and J. M. Vlissides. "InterViews: A C++ Graphical Interface Toolkit". Technical Report CSL-TR-88-358, Stanford University, July 1988.

[MGR91] D. J. Maguire, M. F. Goodchild, and D. W. Rhind. *"Geographic Information Systems"*, volume 1. Longman Scientific & Technical, copublished in the US with John Wiley & Sons, Inc. New York, 1991.

[Mor92] S. Morehouse. "The ARC/INFO Geographic Information System". *Computers and Geosciences: An International Journal*, 18(4), August 1992.

[MP94] C. B. Medeiros and F. Pires. "Databases for GIS". In *SIGMOD Record*, March 1994.

[NHS84] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. "The Grid File: An Adaptable, Symmetric Multikey File Structure". *ACM Transactions on Database Systems*, March 1984.

[Ous90] J. Ousterhout. "Tcl: An embeddable command language". In *Proceedings of the 1990 Winter USENIX Conference*, 1990.

[Ous91] J. Ousterhout. "An X11 toolkit based on the Tcl language". In *Proceedings of the 1991 Winter USENIX Conference*, 1991.

[Rob81] J. T. Robinson. "The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes". In *Proceedings of the 1981 ACM-SIGMOD Conference*, April 1981.

[Sam89] H. Samet. *"The Design and Analysis of Spatial Data Structures"*. Addison-Wesley, 1989.

[Sea92] Michael Shapiro and et. al. *"GRASS 4.0 Programmer's Manual"*. U.S. Army Consrtuction Engineering Research Laboraroty, 1992.

[SFGM93] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. "The SEQUOIA 2000 Storage Benchmark". In *Proceedings of the 1993 ACM-SIGMOD Conference*, Washington, D.C., May 1993.

[SR86] M. Stonebraker and L. A. Rowe. "The Design of Postgres". In *Proceedings of the 1986 ACM-SIGMOD Conference*, 1986.

[Sto86] M. Stonebraker. "The Case for Shared Nothing". *Database Engineering*, 9(1), 1986.

[Sto94] M. Stonebraker, editor. *"Readings in Database Systems"*, pages 492–505. Morgan Kaufmann, 1994.

[SWKH94] M. Stonebraker, E. Wong, P. Kreps, and G. Held. *"The Design and Implementation of INGRES"*, pages 37–53. Morgan Kaufmann, 1994.

[TS87] C. Faloutsos T. Sellis, N. Roussopoulos. "The R^+–Tree: A Dynamic Index for Multi–Dimensional Objects". In *Proceedings of the 13th VLDB Conf.*, September 1987.

[Ube94] M. Ubell. "The Montage Extensible DataBlade Architecture". In *Proceedings of the 1994 ACM-SIGMOD Conference*, May 1994.

[vOV91] P. van Oosterom and T. Vijlbrief. "Building a GIS on Top of the Open DBMS Postgres". In *Proceedings EGIS'91, Second European Conference on Geographical Information Systems*, April 1991.

[VvO92] T. Vijlbrief and P. van Oosterom. "The GEO System: an Extensible GIS". In *Proceedings of the International Symposium on Spatial Data Handling, Charleston, South Carolina*, August 1992.

[Wel84] T.A. Welch. "A Technique for High-Performance Data Compression". *IEEE Computer*, 17(6), 1984.

[Zan83] C. Zaniolo. "The Database Lanaguage GEM". In *Proceedings of the 1983 ACM-SIGMOD Conference*, May 1983.