THE SEQUOIA 2000 STORAGE BENCHMARK

Michael Stonebraker, James Frew, Kenn Gardels and Jeff Meredith
EECS Dept.
University of California, Berkeley

Abstract

This paper presents a benchmark that concisely captures the data base requirements of a collection of Earth Scientists working in the SEQUOIA 2000 project on various aspects of global change research. This benchmark has the novel characteristic that it uses real data sets and real queries that are representative of Earth Science tasks. Because it appears that Earth Science problems are typical of the problems of engineering and scientific DBMS users, we claim that this benchmark represents the needs of this more general community. Also included in the paper are benchmark results for four example DBMSs, ARC-INFO, GRASS, IPW and POSTGRES.

1. INTRODUCTION

There have been numerous benchmarks oriented toward DBMS performance in a variety of application areas. Perhaps the most famous one, TP1 [ANON85] is oriented toward business data processing, and has spawned a collection of derivative benchmarks, the most recent being TPC-A, TPC-B and TPC-C. These benchmarks represent the typical needs of a transaction processing user of a DBMS. They consist of short update-oriented transactions that will stress the transaction system and the basic overhead of simple command processing. Another benchmark [CATT92] is oriented toward electronic computer aided design (ECAD) applications. It contains a set of more complex commands that have high locality of reference on a tiny (main memory) data set, and it stresses the efficiency of a client-server DBMS connection in a very specialized environment (i.e. security can be ignored). An extensive collection of other DBMS-oriented benchmarks is contained in [GRAY91].

We feel that there is a broad application area, namely engineering and scientific data bases, that has special needs not addressed by any of the above benchmarks. This community is typified by Earth Scientists, whose DBMS needs we are trying to support in the SEQUOIA 2000 research project [STON92]. Earth Scientists are usually geographers, hydrologists, oceanographers, or chemists by background, and are united by common problems concerning our survivability on Earth. They investigate issues surrounding global warming, ozone depletion, environment toxification, species extinction, etc.

Loosely speaking, Earth Science research can be divided into three categories:

field studies
remote sensing
simulation

Researchers who perform field studies usually obtain geographic data, typically in data sets of the form:

{ (longitude, latitude, elevation, array-of-values) }

For example, one SEQUOIA 2000 group at the Santa Barbara Campus has collected extensive field data from the Antarctic Ocean about the effect of ozone depletion on ocean organisms [SMIT91]. Such data consist of various ocean characteristics at various depths for specific geographic locations.

This research was sponsored by Digital Equipment Corp. under Research Grant 1243.

Researchers in remote sensing focus on analyzing and interpreting satellite imagery. Such imagery can be thought of as an array of values of the form:

value (longitude, latitude, wavelength band, time)

For example, the Thematic Mapper (TM) sensors on the Landsat satellites sample the Earth's surface on a 30 meter by 30 meter grid, in 7 wavelength bands, repeating every 15 days. For more information on the requirements of remote sensing users, the interested reader is directed to [LOHM83].

Climate modelers use general circulation models (GCMs) for simulating regional or global phenomena. Such models are similar to computational fluid dynamics (CFD) models in that they **tile** the study area and then compute a collection of **state** variables for each tile at time T+1 based on the state in the tile at time T and that of neighboring tiles at time T. The output of such simulation models is an array of values of output parameters such as temperature and barometric pressure as a function of time:

array-of-values (longitude, latitude, elevation, time)

Because GCMs are so computationally intensive, Earth Scientists wish to save **all** GCM simulation output for extensive periods of time. Subsequent analysis and visualization efforts can use these stored data, rather than requiring a model rerun.

The characteristics of the Earth Science (ES) applications we have been discussing are:

1) massive size

ES data bases usually include substantial numbers of images and simulation output, and are extremely large. For example, the four main SEQUOIA 2000 ES research groups collectively would like to store about 10 ** 14 bytes (100 Tbytes) of data. Or consider the NASA Earth Observation System (EOS), a collection of satellites to be launched in the late 1990's to support the needs of the ES community. Collectively, these satellites will send 1 Tbyte of data per day back to ground stations. The ground storage and distribution system (EOS/DIS), currently being built by a government contractor, is charged with storing all EOS data for 15 years. When completed, this data base will be some 10 ** 16 bytes (10 petabytes), and will be the Earth's largest data base.

We see that data base size in the ES community is often much larger than the modest size of TP1 benchmark data bases. The Cattell benchmark is even more modest in its size requirements.

2) complex data types

ES data bases often include multi-dimensional arrays, geometries for spatial objects, and other complex data types. How well any DBMS performs in this environment is largely determined by its support for arrays, spatial objects and complex objects. Such types are rarely present in other benchmarks, which limits their relevance to the ES community.

3) Sophisticated searching

ES data base applications include the requirement of searching arrays and spatial data for desired information. B-trees are rarely adequate for the search needs of this community. On the other hand, most other benchmarks can be adequately addressed by systems relying exclusively on B-tree indexing.

Because other benchmarks do not address the community of users we are trying to support in SEQUOIA 2000, we have chosen to create a new benchmark. The purpose of our benchmark is twofold. First, SEQUOIA 2000 Computer Scientists need a standard baseline case on which to judge new technical ideas. Second, publication of this benchmark should heighten awareness in the DBMS community of ES needs. Although DBMS research has yielded spatial access methods e.g. [ROBI81, NIEV84, GUTM84, FALO87, SAME84, LOME90] and spatial query languages e.g [ROUS85], little of this work has found its way into real general purpose systems. We hope that this benchmark will cause system builders to focus more energy in the direction of ES users.

We have created the following benchmark to abstract the kinds of data SEQUOIA 2000 researchers use and the operations they wish to perform. This benchmark is driven by user needs and is not trying to make any particular DBMS look good or bad. We know of no DBMS that works well on the benchmark in this paper. It should be considered a **target** for DBMS software to strive for.

Although we are specifically targeting this benchmark to the needs of ES users, it appears that it represents the needs of a broad class of engineering and scientific DBMS users. For example, Geographic Information Systems (GIS) users, such as governmental agencies and utility companies, wish to store spatial data from satellite imagery and other sources; they resemble the field studies and remote sensing ES users discussed above. Other engineering and scientific DBMS users include physicists, chemists and biologists doing research at the National Laboratories, and members of the technical staff at industrial firms, such as aerospace and petroleum companies, concerned with engineering applications. Such users typically store large data arrays, and their needs appear to be similar to the simulation users described above. In [BELL88] one can find additional information on the concerns of this class of users.

The remainder of this paper is organized as follows: In Section 2 we present the data base that we use in the benchmark. Section 3 presents the operations that must be performed. Section 4 turns to the environment in which the benchmark should run and the reporting requirements for benchmark results. Section 5 runs the benchmark on four target DBMSs and presents benchmark performance on this collection of DBMSs.

2. BENCHMARK DATA

2.1. Introduction

ES researchers typically focus on problems at the following four **scales:**

local	i.e. a river drainage basin
regional	i.e. a study area of one or two states
national	i.e. a study area of one country
world	the study area is the whole world

In addition, most studies that use remote sensing or simulation data **tile** the study area into rectangular cells, and then record data for each tile. There are three popular granularities for tile size:

coarse	tile size of several kilometers; typical of simulation output.
medium	tile size of one kilometer; typical for studies using the Advanced Very High Resolution Radiometer (AVHRR) sensor data.
fine	tile size less than 100 meters, typical for studies using Thematic Mapper (TM) sensor data.

The data set size for any particular study depends crucially the scale of the problem and the granularity of the data, and this can vary over many orders of magnitude. To capture this diversity, we propose three different benchmarks, each of which contains the same data for a different size study area, as follows:

regional benchmark:

This benchmark typifies the needs of an Earth Scientist working on a **regional** problem, such as vegetation classification in the state of California. The geographic region in the benchmark is a 1280 km x 800 km rectangle encompassing the states of California and Nevada.

national benchmark:

This benchmark typifies the needs of an Earth Scientist working on a problem of national scale. The benchmark geographic region is a rectangle covering the United States that is 5500 km x 3000 km.

world benchmark:

Some Earth Scientists study problems on the scale of the entire earth. The final benchmark study area is the entire globe.

Because we are distributing real Earth Science data for this benchmark to any interested parties, we focus on the regional and national benchmarks, which can be feasibly sent to a researcher on 8mm tapes. Distribution of the world data set will have to await better tape densities. In this paper we do not precisely define the largest data set, but leave it for a future exercise.

ES researchers deal with four kinds of data routinely in their work:

raster data
point data
polygon data
directed graph data

Our benchmark includes a representative of each class as follows:

Raster Data

We have chosen to include data from the Advanced Very High Resolution Radiometer (AVHRR) sensor on the NOAA satellites. This sensor decomposes the entire earth into 1.0 km x 1.0 km **tiles.** As distributed on CD-ROM by the United States Geological Survey (USGS), these tiles are aligned with a Lambert Azimuthal Equal Area map projection. For the regional data set, each geographic point is represented by a pair of 32 bit integers representing respectively the distance in meters that the point is east and north of an origin located at 100 degrees West longitude, 45 degrees North latitude.

For each tile, five onboard sensors capture 10-bit values corresponding to the energy observed in each of five wavelength bands. The satellite observes each tile twice per day; however the USGS publishes the data every two weeks, using values from a composite of passes that ensures that the data for each tile is from a cloud-free observation. Our benchmark therefore contains 26 data sets, corresponding to 52 weeks of elapsed time.

Some satellites have a medium tile size close to that of AVHRR, while others have a fine tile size, e.g. Thematic Mapper (TM) which uses 30 meter tiles. To have a single data set that best represents the general characteristics of satellite data sets, we have chosen to reduce the tile size of AVHRR to 0.5 km x 0.5 km. To accomplish this reduction, we have **oversampled** AVHRR data to produces the desired factor of 4 data expansion.

The regional version of the benchmark therefore contains

2 x 1280 x 2 x 800 = 4,096,000 tiles

for each of which we record

(5 observations) * (2 bytes) = 10 bytes

The 26 observations over 1 year thereby constitute 1.064 Gbytes of data.

The national data set has the same data as the regional data set for an area that is 16.1 times as large, so it is about 17 Gbytes. The world data set covers an area 100 times as large and is nearly 2 Tbytes.

Point Data

For the study region, we include the names and locations of specific geographic features that have a point location. This data set, available from the USGS Geographic Names Information System (GNIS), has been reprocessed into the same Lambert Azimuthal Equal Area projection noted above. Specifically, the data are a collection of character string names, each associated with a {distance-north-of-origin, distance-east-of-origin} pair of 32 bit integers. There are 76,584 entries in the regional benchmark, and each name is a variable length string, with average length of 16 bytes. Since each geographic point is represented as a pair of 32 bit integers, this data set is a collection of records with an average length of 24 bytes. The national benchmark has about 15 times as many points. The sizes of the point data sets are about:

regional benchmark: 1.83 Mbytes
national benchmark: 27.5 Mbytes

Polygon Data

For polygonal data we have chosen to use a data set consisting of regions of homogeneous landuse/landcover, available from USGS. Again, we have chosen to reprocess the data into the same coordinate system as the point and raster data. Each record in this data set is a polygon, consisting of a variable number of points, represented as pairs of integers, together with an integer encoding for one of 37 landuse/landcover type. Since the average polygon has 50 sides, this data set has an average record size of 204 bytes. There are 93,607 polygons in the regional benchmark and about 1.4 million polygons in the national benchmark. The sizes of the polygon data sets are about:

regional benchmark: 19.1 Mbytes
national benchmark: 286 Mbytes

Directed Graph Data

Our last data set represents information for a directed graph. Here we have chosen to use another USGS data set containing information about drainage networks. Here, each river is represented as a collection of **segments.** Each segment is represented as a non-closed polygon with a beginning node and ending node. For each segment, the data set records segment geometry and the segment identifier. For the regional benchmark, there are 286,300 segments, and for the national benchmark there are about 6.5 million segments. The data sets sizes are:

regional benchmark: 47.8 Mbytes
national benchmark: 1.1 Gbytes

The complete regional data set is a little over 1 Gigabyte and is well suited to deployment on a disk storage system. The national benchmark is around 18 Gigabytes. Although it is possible to buy enough disk space to hold this benchmark, the intention of the designers is that this be a tertiary memory data set on either an optical disk robot or a tape robot. The world benchmark is multiple terabytes, and can only be deployed on the largest of current commercial tape robot devices.

Over the rest of this decade as hardware prices decline, we expect the regional benchmark will move to a main memory data set, the national benchmark to a disk data set and the world benchmark to an ‘‘easy’’ tertiary memory data set. We have purposely designed the benchmark for **durability** in the face of technological change. Many other benchmarks, become quickly obsolete because of increasing storage capacities at all levels of a storage system. For example, the Wisconsin benchmark [BITT83], originally designed as a disk benchmark, now fits easily into most common main memory caches, thereby dramatically reducing its utility.

3. THE BENCHMARK QUERIES

In this section we present the 11 queries that form the benchmark. In each case, we write the query in words and also express it in POSTQUEL [MOSH92]. The POSTQUEL queries assume the following POSTGRES schema:

```
create RASTER (time = int4, location = box, band = int4, data = int2[][])
create POINT (name = char[], location = point)
create POLYGON (landuse = int4, location = polygon)
create GRAPH (identifier = int4, segment = open-polygon)
```

In this schema, we use the point, open-polygon, box, and polygon data types with the obvious interpretation. Each is internally represented as a collection of points, represented as a pair of integers. For each of the five frequencies, the AVHRR data are logically a giant two-dimensional array. We have chosen to include the possibility of **chunking** this array into smaller subarrays for storage convenience. The data type int2[][] stores the AVHRR data elements for each location in the subarray, and the location field in the RASTER class gives the bounding rectangle for each subarray. Collectively, the subarrays cover the entire study region. For the regional benchmark, chunking is not required, as each array is 8 Mbytes. Since each array in the national benchmark is 129 Mbytes, chunking is a desirable feature.

Users are free to use any schema they wish, as long as AVHRR elements are 16 bit objects and geographic points are pairs of 32 bit objects. Also, users are free to decompose each AVHRR image into multiple subimages, as we have indicated above, if that suits their needs better.

The remainder of this section presents the 11 benchmark queries. They are grouped into five collections:

- data load
- raster queries
- polygon and point queries
- spatial joins involving one or more types of objects
- recursion

The queries use the following constants:

RECTANGLE: a geographic rectangle of size 100km x 100km randomly placed in the study region
BAND: a random wavelength band
TIME: a random time
LANDUSE: a random landuse/landcover type
LOCATION: a random geographic point in the study area
POINT-NAME: the name of a random point in the POINT class
INT-1: an integer, set for this benchmark at 64
FLOAT-1: a floating point number, set for this benchmark at 1.0
FLOAT-2: a floating point number, set for this benchmark at 10.0

For each task we show the code that is required in the query language, POSTQUEL [MOSH92] for the schema discussed earlier. This is done to assist the reader in understanding the semantics of the operation.

3.1. Data Load

Query 1: Create and load the data base and build any necessary secondary indexes.

Earth Scientists expend much effort loading new data into their computer systems. This activity, usually disregarded in other benchmarks, is included as Query 1. It is common knowledge that commercial systems differ by as much as a factor of 10 in the speed with which they can load data and build indexes. In many cases the low performance systems enter data by running one insert query per record while the high performance ones have a streaming ‘‘bulk’’ copy facility that interacts with a lower level of the DBMS. Query 1 will expose such performance differences.

It is appropriate to begin timing Query 1, after the tape containing the benchmark has been copied to disk. Otherwise, Query 1 will presumably run at the speed of the tape reader. Once the data have been copied to disk, timing for Query 1 should record the elapsed time to load the data into the system being tested, performing whatever data conversions are desired, and building any secondary indexes. For the medium benchmark, Query 1 can be broken into a collection of **partial** loads, if it is more convenient that way.

For POSTGRES we implemented the schema discussed earlier and used the standard copy utility included in the system. We chose to build R-tree [GUTM84] indexes on location in the POINT and POLYGON classes and on segment in the GRAPH class. We also require B-tree indexes on time and band for RASTER, name for POINT, and name for POLYGON. Lastly, we use an index on the function, size, operating on polygon locations in one query. The POSTGRES timing for Query 1 includes the time to run the copy command for the four input data sets and the time to build the 8 indexes noted above.

3.2. Raster Queries

Earth Scientists who deal with satellite data run many queries on raster data. This section presents examples of three of their common queries. Query 2 is a **time travel** query, namely:

Query 2: Select AVHRR data for a given wavelength band and rectangular region ordered by ascending time.

```
retrieve  (clip (RASTER.data, RECTANGLE), RASTER.time)
where RASTER.band = BAND
```

```
order by ascending time
```

Here, the two constants are run-time parameters that denote respectively a rectangle corresponding to the desired geographic rectangle and the wavelength band required. The query then returns 26 raster images for the appropriate wavelength band, each clipped to the correct rectangle, ordered by ascending calendar time. In effect, the user wants to play a time-travel movie of the images to watch what happens to the study rectangle as time increased. The 26 images are each 80K bytes, so the query returns about 2.1 Mbytes of data to the application program.

Timing for this query must include returning the data to an application program; however, the application need not put the data on the screen. We are attempting only to test the storage and retrieval components of a system and not the visualization software that displays results. This topic is the subject of a separate benchmark, currently under construction [OLSO92].

The third query performs a **spectral analysis** as follows:

Query 3: Select AVHRR data for a given time and geographic rectangle and then calculate an arithmetic function of the five wavelength band values for each cell in the study rectangle.

```
retrieve  (raster-avg {clip (RASTER.data), RECTANGLE})
where RASTER.time = TIME
```

Here, raster-avg is a user-defined function that computes a weighted average of the individual cell values in RASTER.data. The intent of this query is for the function applied, e.g. sum, average, sum over restricted frequencies, etc. be a run-time parameter – it is not allowed for the person performing the benchmark to precompute the answer to this query during the execution of Query 1. The reason for this restriction is that Earth Scientists typically run many different weighted averages for a given study area, looking for the one that produces the best output.

The fourth query changes the spatial resolution of a raster image.

Query 4: Select AVHRR data for a given time, wavelength band, and geographic rectangle. Lower the resolution of the image by a factor of 64 to a cell size of 4km x 4km and store it as a new DBMS object.

```
retrieve into FOO-1 (
    time = RASTER.time,
    location = RASTER.location,
    band = RASTER.band,
    data = lower-res ( clip (RASTER.data, RECTANGLE), INT-1))

where RASTER.time = TIME
and RASTER.band = BAND
```

Here, RECTANGLE is a rectangle corresponding to the viewing region of interest. INT-1 specifies the amount of resolution reduction, here 64, and BAND and TIME give the wavelength band and time of interest.

This operation is useful in creating **abstracts** of raster data. Earth Scientists need to browse through massive amounts of data, and it is useful for them to see much of it at low resolution and then **zoom** into areas of particular interest. Hence, many scientists wish to have raster data at multiple levels of detail.

3.3. Point and Polygon Queries

Data obtained from field studies are often about geographic points or polygons. Many researchers also classify raster data into polygons that have a common characteristic (e.g. land use, snow cover). It is natural to have queries for these kinds of objects and there are three in our benchmark.

The first one is a conventional **non-spatial subsetting** of POINT data on a non-spatial attribute, namely:

Query 5: Find the POINT record that has a specific name.

```
retrieve (POINT.all)
```

```
where POINT.name = POINT-NAME
```

To satisfy this query, a system must have some sort of non-spatial indexing (B-tree, hashing, etc.) and be able to assemble spatial and non-spatial attributes for output.

The next query performs a natural **spatial subsetting** operation on the point data.

Query 6: Find all the polygons that intersect a specific rectangle and store them in the DBMS.

```
retrieve into FOO-2 (POLYGON.all)
where POLYGON.location || RECTANGLE
```

Here, || is a user-defined ‘‘polygon intersects rectangle’’ operator that returns true if the location of the polygon intersects the rectangle specified by RECTANGLE.

This query requires a spatial index of some sort, and will be difficult to execute efficiently on a system that only supports B-trees. Like Query 4, it requires the ability to dynamically create new data base tables or classes, a property not found in all DBMSs.

The last query is a **combination** query that has both spatial and non-spatial restrictions.

Query 7: Find all polygons that are more than a specific size and within a specific circle.

```
retrieve (POLYGON.all)
where size (POLYGON.location) > FLOAT-1
and POLYGON.location <|> circle (LOCATION, FLOAT-2)
```

Here, FLOAT-1 is the threshold polygon size, set for this benchmark at 1 square km, while LOCATION and FLOAT-2 define a circle. Specifically, LOCATION is the center and FLOAT-2 is the radius, set for this benchmark at 10 km. The operator, <|>, returns true if the polygon which is the left operand is inside the circle which is the right operand. Efficient execution of this query requires a query optimizer that can evaluate the expected selectivity of both the spatial and the non-spatial clause, and then choose the more restrictive one to evaluate first. For the benchmark data most polygons are larger than 1 square km, so the clause that spatially subsets the data should be preferentially used.

3.4. Spatial Joins

Many Earth Scientists require the ability to correlate (or join) multiple kinds of data. In this section we present three benchmark queries that join data of one spatial type to those of a different spatial type.

Query 8 finds the polygons that intersect a rectangle of interest. The rectangle is defined to have a center which is a named geographic point of interest. This query performs a complex spatial join of the POINT and POLYGON data sets.

Query 8: Show the landuse/landcover in a 50 km quadrangle surrounding a given point.

```
retrieve (POLYGON.landuse, POLYGON.location)
where POLYGON.location || make-box (POINT.location, 50)
and POINT.name = "POINT-NAME"
```

This query finds all polygons that intersect the rectangle of interest. POINT-NAME is the name of the point that lies at the center of a 50 km by 50 km rectangle of interest. The function make-box creates this rectangle from the center point and the length of each side. The operator || returns true if a polygon intersects the rectangle of interest.

Query 9 performs a join between raster data and polygon data.

Query 9: Find the raster data for a given landuse type in a study rectangle for a given wavelength band and time.

```
retrieve (POLYGON.location, clip (RASTER.data, POLYGON.location))
where POLYGON.landuse = LANDUSE
and RASTER.band = BAND
and RASTER.time = TIME
```

Here, LANDUSE gives the landuse classification that is desired, while BAND and TIME specify the wavelength band and time of the desired raster data. The join is implicit in the arguments of the clip function.

The last query is a join between point and polygon data as follows:

Query 10: Find the names of all points within polygons of a specific vegetation type and create this as a new DBMS object.

```
retrieve into FOO-3 (POINT.name)
where POINT.location || POLYGON.location
and POLYGON.landuse = LANDUSE
```

The operator **||** is the ‘‘point inside polygon’’ operator.

Note that in this section, the meaning of **||** has been context-sensitive. POSTGRES allows tokens like **||** to be **overloaded,** so they can be used to mean different things for different operand types.

3.5. Recursion

Earth Scientists often want to trace drainage basins or irrigation networks. This involves restricted recursive queries on network data. Our last query embodies this sort of activity, and is termed the **Dunsmuir spill query** after an incident during 1991 in which a Southern Pacific freight train derailed and spilled toxic chemicals into the Sacramento River near the town of Dunsmuir, California. After such an incident, an Earth Scientist would like to find all the waterways into which the spilled chemicals could flow. For naturally occurring waterways, this answer is usually ‘‘downstream’’ in the same waterway. However, waterway data in California often represent irrigation networks, where there may be many places downstream from a given point.

Query 11: Find all segments of any waterway that are within 20 km downstream of a specific geographic point.

```
retrieve into temp (GRAPH.identifier, GRAPH.segment, partial-length (GRAPH.segment, LOCATION))
where LOCATION *&* GRAPH.segment

append* to temp (GRAPH.identifier, GRAPH.segment, length = temp.length + length (GRAPH.segment)
where end(temp.segment) = begin (GRAPH.segment)
and GRAPH.identifier notin {temp.identifier}
and temp.length < 20
```

Here, the first query identifies the segment on which the initial spill point, LOCATION, is located. It also, calculates the distance from the spill point to the end of the segment in the function, partial-length. The operator *&* returns true if a point is on a specific segment. The second append command runs an indefinite number of times, signified by the *, and stops when no new segments get added in an iteration. Each iteration adds one or more segments and the distance they are from the initial spill point. The iteration ceases when all segments are more than 20 km from the spill point.

This command should be taken as an indication of the kinds of recursive queries that real scientists wish to run. The interested reader should note that computation is required in the middle of the recursion, that the scope of the recursion is relatively small, and that the search space can be radically pruned at the beginning (for example by eliminating all segments more than 20 km from the spill point). In fact, the technique used to perform the recursion is much less important than the utilization of input pruning. Optimizing this query will require different capabilities than available in current systems with recursive processing, such as LDL [CHIM91] and NAIL [ULLM85].

4. BENCHMARK CONSTRAINTS AND REPORTING CONVENTIONS

There are several points that we wish to make in this section. First, it is permissible to run the benchmark on any combination of hardware and software that the user desires. The result of the benchmark should be reported as a collection of 11 numbers indicating the elapsed time for each task. The retail price of the hardware on which the benchmark is run should also be reported. If the entire benchmark can be run, then a single overall performance number indicating elapsed time per unit hardware cost should be reported:

performance = (total elapsed time for the benchmark) / (retail price of hardware)

In this way, the benchmark can be run on any machine from a PC to a supercomputer.

If users can only run part of the benchmark, either because their systems are not powerful enough to express the other tasks or because the programming of the task would be too difficult to accomplish, then they should report the results for the queries that could be run.

Second, the result of each query in the benchmark is either a new object stored in the data base (Queries 4, 6 and 9), or the appropriate data returned to the application program, which can discard them. As noted earlier, there is no requirement that the data be displayed on the screen − this is a **storage** benchmark, not a **visualization** benchmark.

Third, the benchmark can be coded in any language appropriate for performing this task. For example, to run the benchmark against a RDBMS, then it should be transliterated into SQL. To run against an OODBMS, it should be recast in the query language of the particular system. If the query is run using some low level algorithmic interface to a DBMS, then numbers must also be reported for the same task performed using the high-level declarative interface.

Finally, Earth Scientists fall somewhere in a middle ground regarding concern for security. On the one hand, they are not as security conscious as applications that store financial data; however, they grimace at any system with no security. A package that executes the DBMS in the same protection domain as the application program will not fulfill the minimum needs of this community. Therefore, any system that does not run the application in a separate protection domain from the DBMS must clearly note this fact.

5. BENCHMARK RESULTS

It would be natural to run our benchmark on one of the popular commercial relational DBMSs. Since none offer a spatial access method or support for arrays, we would have to simulate these features. To use B-trees for spatial indexing, we would have to transform two-dimensional spatial data into a one-dimensional structure suitable for B-tree indexing. Z transforms [OREN86] are one of several techniques that could accomplish this task. However, Queries 6, 7, 8, 9, and 10 deal with spatial areas and not points. To solve any of these queries one must investigate multiple intervals in the one-dimensional space, thereby slowing performance.

To simulate arrays, we could simply use the binary large objects (blobs) present in many DBMSs. However, the only operations available for blobs are storing and retrieving them, and all raster operations would have to be programmed with user space code. Moreover, the spatial joins in Queries 8, 9, and 10 would have to be programmed by executing some sort of join strategy implemented within an application program. This would entail a fair amount of programming, as well as being very slow.

Rather than engaging in a lot of programming to produce extremely poor results, we have elected not to test this class of products. For the same reasons, we have omitted current object-oriented DBMSs, all of which lack a spatial access method and array support.

Instead, in this section we focus on four systems that offer support of one sort or another for spatial objects. These are

ARC-INFO: a popular geographic information system marketed by Environmental Systems Research Institute of Redlands, Ca.

GRASS: a public domain geographic information system written by the Center for Environmental Research and Languages (CERL) at the US Army Corps of Engineers Construction Engineering Research Laboratory in Champaign Illinois.

IPW: A raster-oriented image processing package written at the University of

California, Santa Barbara [FREW90].

POSTGRES: a next generation DBMS prototype written at the University of California, Berkeley.

In Figure 1, we report the results for the regional benchmark. The first three systems were run on a DECsystem 5900 with 2 Gbytes of disk space and 128 Mbytes of main memory which retails for $67,300. The fourth was run on a SUN ipx with 32 Mbytes of main memory and 2 Gbytes of disk space, retailing for $12,000. The Sun machine is about 2/3 the speed of the DEC machine with a comparable performance disk system.

The POSTGRES benchmark was executed on Version 4.0 by loading the data into a data base consisting of the schema from Section 3, using the POSTGRES copy utility. Then, the queries that appeared in the text of the previous section were run, except Query 11. Although the recursion operator (*) was available in an earlier version of POSTGRES, the support code was buggy, and it is not available in the current release. The recursion would have to have been programmed in user code, a solution we felt violated the spirit of the benchmark.

Also, the R-tree access currently has a bug for the polygon data that caused POSTGRES to crash on Queries 6, 7 and 8. Rather than delay this report, we have chosen to report numbers do not benefit from R-tree search on POLYGON.location. With R-tree indexing all three of these queries will take at most a few seconds.

IPW consists of a collection of UNIX shell commands that manipulate raster images, assumed to be stored one per file, in a specific format. These facilities allow IPW to perform the required data load and queries 2-4. The remainder of the queries are extremely difficult or impossible using IPW, so no loading of the point or polygon data was attempted. AVHRR data were loaded, one file per wavelength band per time period, and the file name connoted this fact. Query 2 was accomplished by specifying the file containing the correct wavelength band and time and then using the IPW command that subsets a raster image. Query 3 was performed by assembling the correct collection of subsetted images and then combining them with the IPW weighted average command. Query 4 was accomplished by the windowing operation followed by a subsampling operation.

As required in the benchmark reporting section, it should be carefully noted that IPW does not meet the minimum security requirements of the benchmark.

The basic GRASS unit of manipulation is a **map,** which can be either in raster or vector format. Raster maps are stored one per UNIX file in a standard array representation, with a header containing assorted information about the map. Vector maps are also stored one per file with points, lines and polygons encoded in a straightforward way.

Queries 2-4 can be performed using GRASS raster capabilities. Specifically, Query 2 is performed by identifying the correct map manually, and then clipping the map to the correct size. Query 3 is performed using a clip followed by a weighted average. Finally, Query 4 is accomplished using the GRASS clip and subsampling commands. Since GRASS has no notion of character string attributes, Query 5 cannot be performed.

Even though GRASS supports vector maps, it has no query facilities for them, so queries 6-7 are difficult to perform. To get a correct answer, the point and polygon data were converted to raster format: a grid was superimposed on top of the vector data, and a value indicating the polygon identifier was recorded in each raster cell that lay within the polygon. A similar technique was used for the point data. In both cases, the technique will not extend to data sets containing overlapping polygons. Queries 6 and 7 were then accomplished by creating a raster map containing the correct subsetting region, using the technique above, and then intersecting the data map with the subsetting map. One reason the load time is so long for GRASS is that the polygon data and point data are stored redundantly in raster format at load time. Although this technique accomplishes the given task, it is clearly an obtuse way to compensate for missing capabilities in GRASS.

As required in the benchmark reporting section, it should be carefully noted that GRASS does not meet the minimum security requirements of the benchmark.

Like GRASS, ARC-INFO also contains support for both vector and raster data. In addition, it can store related **attribute** data of assorted types. Vector maps are stored by decomposing polygons into a collection of **arcs,** which are then indexed using a proprietary data structure. Points are considered as lines with zero length and are indexed using the same data structure. Attribute data are loaded into a separate data manager, INFO, which is a primitive, quasi-relational DBMS. Coupling spatial and attribute data requires, in effect, a join between data in these two different systems, a task accomplished by special ARC-INFO code. The load time for ARC-INFO is more than 10 hours, and most of the time is consumed in building the arc-node representation. The build utility seemingly loads about 3-4 polygons per second, very poor performance on a high speed machine. Also, ARC-INFO must store the attribute data twice to give good performance on the benchmark. Specifically, ARC requires that attribute data in INFO be sorted on an internal unique identifier. In this way, attribute data corresponding to a given ARC spatial object can be found using a binary search. Since there are no indexes in INFO, the only way to get fast performance on Query 5 is to store the attribute data a second time sorted on name. Without redundant storage, Query 5 runs for 16 seconds.

Unfortunately, the University of California does not have a license for the raster portion of ARC-INFO. As a result, Queries 2-4 and 9 could not be run, and the time reported for Query 1 does not include loading raster data.

As required in the benchmark reporting section, it should be carefully noted that ARC-INFO does not meet the minimum security requirements of the benchmark.

6. CONCLUSIONS

There are several points that should be noted about this benchmark. First, there are at least two reasonable ways to represent polygon data. First, complete polygons can be stored as noted in the schema of Section 3. Alternatively, polygons can be decomposed into a collection of nodes and arcs, which are then stored. In this case, all **nodes** are found that are on the boundary of more than two polygons. These nodes are connected by an ordered collection of line segments, called **arcs.** Each polygon is represented as a

Query	ARC-INFO	GRASS	IPW	POSTGRES
1	38220*	46260	1530*	5270
2	X	74.0	18.9	28.7
3	X	117	6.0	16.0
4	X	4.0	0.7	6.0
5	1.0	X	X	1.0
6	233	1.0	X	31.0**
7	146	13.0	X	42.7**
8	105	20.0	X	110**
9	X	5.0	X	2
10	2257	380	X	624
11	X	X	X	X
cost	67,300	67,300	67,300	12,000

* - partial load only; see the text for details

** - does not use R-tree search; see the text for details

Benchmark Results in Seconds

Figure 1

collection of arc identifiers.

Using an arc-node format, each arc is stored only once while in the complete polygon representation it is stored twice. This data redundancy requires extra space in addition to presenting a data integrity problem. Namely, whenever an arc is changed in the complete polygon representation, the system must ensure that both copies of it are changed appropriately.

The disadvantage of arc-node format is that it is slower than complete polygon format because of the added indirection required in the polygon queries. POSTGRES, using the complete polygon representation, handily beats ARC-INFO, which uses the arc-node format. This is not too surprising, since most polygon data are read much more often than they are written, and the complete representation is optimal for reading. POSTGRES also requires less space for the polygon data than either ARC-INFO or GRASS, in spite of the space-efficiency of the arc-node format (although any space penalty imposed by a particular polygon format would be insignificant in this benchmark, because the polygon data are a small fraction of the overall data set.) POSTGRES handles the data integrity problem with a collection of DBMS triggers.

The second point to be made about the benchmark data is the number of sides in a polygon. The benchmark contains one polygon with 5184 nodes, and each system must be prepared to deal with polygons with a vast number of sides. Any system that limits objects to 4K or even 8K will have a problem with these data.

Third, the fastest system on raster data is IPW. It is a carefully tailored set of UNIX routines, hand optimized for efficiency; a low-function, high performance (‘‘lean and mean’’) alternative. Its performance advantage relative to GRASS is primarily because it clips images by carefully reading only the data that will qualify. GRASS, on the other hand, reads an image in its entirety to create the clipped region. IPW also beats POSTGRES, but for a different reason. Namely, IPW is careful to pipeline data between functions in Queries 3 and 4, whereas POSTGRES writes temporary data to disk between the two function invocations.

Fourth, notice that rasterizing polygon data, as GRASS does, offers superb performance. Unfortunately, this technique will not accommodate overlapping polygons. Fortunately, our benchmark does not contain any, so this special technique will work correctly.

Fifth, notice that POSTGRES used a B-tree search on size of polygon in Query 7. This yields a time worse than that of Query 6, where a sequential search over POLYGON is performed. In Query 7 the POSTGRES optimizer incorrectly chooses a non-clustered index lookup rather than a sequential search. Moreover, as noted earlier, the optimal query would use the spatial access method anyway. Query 7 illustrates some of the challenges faced by an optimizer on this benchmark.

Lastly, the landuse data in this benchmark have a ‘‘Swiss cheese’’ characteristic. For example, Central Park is a polygon of landuse type ‘‘park’’ completely contained in another polygon, New York City, classified as ‘‘city.’’ The polygons in our data set have an arbitrary number of such ‘‘holes’’ in them. The current POSTGRES implementation of polygons does not support ‘‘Swiss cheese polygons,’’ so Query 10 could not coded as mentioned in Section 3. Instead, a separate class of HOLES had to be created, and the query expanded to perform a three-way join, checking that a point of interest was not in a hole. Obviously, this leads to the poor performance by POSTGRES on Query 10, and the POSTGRES polygon type should be extended to support holes.

REFERENCES

[ANON85] Anon et. al., ‘‘A Measure of Transaction Processing Power,’’ Tandem Technical Report 85.1, Tandem Computers, Cupertino, Ca., August 1985.

[BELL88] Bell, J., ‘‘A Specialized Data Management System for Parallel Execution of Particle Physics Codes,’’ Proc. 1988 ACM-SIGMOD Conference on Management of Data, Chicago, Il., June 1988.

[BITT83] Bitton, D. et. al., ‘‘Benchmarking Database Systems, A Systematic Approach,’’ Proc. 1983 VLDB Conference, Florence, Italy, November 1983.

[CATT92] Cattell, R. and Skeen, J., ''Object Operations Benchmark,'' ACM Transactions on Database Systems, March 1992.

[CHIM91] Chimenti, D. et.al., ''The LDL System Prototype,'' IEEE Transactions on Knowledge and Data Engineering, March 1991.

[FALO87] Faloutsos, C. et. al., ''Analysis of Object Oriented Spatial Access Methods,'' Proc. 1987 ACM-SIGMOD Conference on Management of Data, San Francisco, Ca., May 1987.

[GRAY91] Gray, J., ''The Benchmark Handbook,'' Morgen-Kaufman, San Mateo, Ca., 1991.

[GUTM84] Gutman, A., ''R-trees: A Dynamic Index Structure for Spatial Searching,'' Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass. June 1984.

[FREW90] Frew, J., ''The Image Processing Workbench,'' Ph.D. dissertation, Dept. of Geography, University of California, Santa Barbara, 1990.

[LOHM83] Lohman, G. et. al., ''Remotely Sensed Geophysical Databases: Experiences and Implications for Generalized DBMS,'' Proc. 1983 ACM-SIGMOD Conference on Management of Data, San Jose, Ca., May 1983.

[LOME90] Lomet, D. and Salzberg, B., ''The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance,'' ACM Trans on Database Systems 15,4 (Dec 1990) 625-658.

[MOSH92] Mosher, C., ''The POSTGRES Version 4.0 Reference Manual,'' Electronics Research Laboratory, University of California, Berkeley, Ca., July 1992.

[NIEV84] Nievergelt, J. et.al., ''The Grid File: An Adaptable, Symmetric Multikey, File Structure,'' ACM-TODS, January, 1984.

[OLSO92] Olson, M. et. al., ''The SEQUOIA 2000 Visualization Benchmark,'' (in preparation)

[OREN86] Orenstein, J., ''Spatial Query Processing in an Object-oriented Database System,'' Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

[ROBI81] Robinson, J., ''The K-D-B Tree: A Search Structure for Large Multidimensional Indexes,'' Proc. 1981 ACM-SIGMOD Conference on Management of Data, Ann Arbor, Mich., May 1981.

[ROUS85] Rousoupoulos, N. and Leifker, D., ''Direct Spatial Search on Pictorial Databases Using Packed R-trees,'' Proc 1985 ACM-SIGMOD Conference on Management of Data, Austin, Tx., June 1985.

[SAME84] Samet, H., ''The Quadtree and Related Hierarchical Data Structures,'' ACM Computing Surveys, June 1984.

[SMIT91] Smith, R. C., et al., ''Optical Variability and Pigment Biomass in the Sargasso Sea as Determined Using Deep Sea Optical Mooring Data,'' Journal of Geophysical Research, September 1991.

[STON92] Stonebraker, M. and Dozier, J., ''SEQUOIA 2000: Large Capacity Object Servers to Support Global Change Research,'' SEQUOIA 2000 Technical Report No 1, Electronics Research Lab, March 1992.

[ULLM85] Ullman, J., ''Implementation of Logical Query Languages for Data Bases,'' ACM-TODS, Sept. 1985.