

# VESPA: A Benchmark For Vector Spatial Databases

Norman W. Paton<sup>1</sup>, M. Howard Williams, Kosmas Dietrich, Olive Liew,  
Andrew Dinn and Alan Patrick

Department of Computing and Electrical Engineering  
Heriot-Watt University, Edinburgh, UK  
email: howard@cee.hw.ac.uk

Department of Computer Science<sup>1</sup>  
University of Manchester, Oxford Road, Manchester, UK  
email: norm@cs.man.ac.uk

**Abstract.** Facilities for the storage and analysis of large quantities of spatial data are important to many applications, and are central to geographic information systems. This has given rise to a range of proposals for spatial data models and software architectures that allow database systems to be used cleanly and efficiently with spatial data. However, although many spatial database systems have been built, there have been few systematic comparisons of the functionality or the performance of such systems. This is probably at least partly due to the lack of a widely used, standard spatial database benchmark. This paper presents a benchmark for vector spatial databases that covers a range of typical GIS functions, and shows how the benchmark has been implemented in two systems: the object-relational database PostgreSQL, and the deductive object-oriented database ROCK & ROLL extended to support the ROSE algebra. The benchmark serves both to evaluate the facilities provided by the systems and to allow conclusions to be drawn on the efficiency of their spatial storage managers.

## 1 Introduction

Benchmarks are useful things. There are two principal kinds of benchmark: *functionality* benchmarks, which are targeted at the question ‘what can this system do?’; and *performance* benchmarks, which seek to answer the question ‘how fast is this system?’. In the absence of benchmarks, making detailed comparisons between alternative proposals concerning their functionality or performance is generally difficult. An overview of database benchmarks is provided by [6].

The authors became aware of the need for a suitable benchmark for vector-based spatial databases at an early stage in their design of a spatial extension to the ROCK & ROLL deductive object-oriented database [3, 9]. At this point it was important to identify a suitable benchmark program that could be used both to test the functionality of the ROCK & ROLL system and to allow performance comparisons with alternative approaches. Unfortunately, no existing publicly

available benchmark could be found that was suitable for use with a vector-based spatial database targeted towards spatial querying and analysis.

The best known existing benchmark for spatial databases is SEQUOIA 2000 [20], which has been used in a number of settings [10, 4, 18]. The SEQUOIA benchmark is explicitly an earth sciences benchmark, and is highly oriented towards raster data. This is reasonable, but makes SEQUOIA inappropriate for use with vector spatial databases. In addition, as SEQUOIA operates with real data sets, it is more difficult to experiment with different data set sizes, as the one data set currently supplied for SEQUOIA stands at around 1Gb. Another performance analysis is provided by [2], but this is more oriented around database population than the time taken for analyses. In some recent work, there has been an emphasis on the generation of spatial data sets (e.g. [11]), but this work has not been followed through into the proposal of complete benchmarks. Other proposals have been narrower in scope than a complete benchmark, for example focusing on spatial join algorithms [14] or indexing [23].

A benchmark which is required to test functionality and assess performance for vector-based spatial databases should ideally satisfy the following requirements:

1. *Ease of use*: Implementation of the data structures and queries of the benchmark should not be particularly time consuming, and obtaining data sets should be straightforward.
2. *Wide ranging functionality*: The benchmark should assess as large a range of spatial database features as possible.
3. *Scalability*: It should be possible to generate data sets that are as small or as large as is required to carry out the performance analyses that are relevant to a particular context.

Due to the absence of any suitable existing benchmark, the benchmark presented in this paper was developed. In terms of the above criteria, it can be seen to satisfy these as follows: *Ease of use* – it has a compact schema, and data for testing purposes is generated synthetically; *Wide ranging* – it has a large number of small test tasks for which performance can be measured; *Scalable* – the data generation software can be used to construct databases that are of a wide range of sizes, from very small, main memory data sets to very large data sets that simulate large scale maps covering a large area. To date, we have used the benchmark to test the functionality and performance of the GIS ARC/INFO [15], the persistent C++ system Exodus [5], the object-relational database PostgreSQL [24], the deductive object-oriented database ROCK & ROLL without built-in spatial extensions [1], and ROCK & ROLL with built-in spatial extensions. The above experiences have shown that the benchmark can be constructed quite rapidly on a range of platforms, although the features it tests can be difficult to support in some environments (e.g. some of the queries did not yield natural implementations in the map-layer oriented ARC/INFO).

The remainder of this paper is structured as follows. Section 2 presents the benchmark. Sections 3 and 4 describe how the benchmark has been implemented

in PostgreSQL and ROCK & ROLL, respectively. Section 5 presents and comments on the results of the benchmark, and section 6 presents some overall conclusions.

## 2 The Benchmark

This section describes the VESPA (VEctor SPAtial) benchmark by describing the data used in the benchmark and the tests that are used to assess the performance of systems.

### 2.1 Data

Unlike the SEQUOIA 2000 [19] benchmark, VESPA uses synthetic data generated by programs written for that purpose. There are several advantages to this approach: obtaining the data is easier, there is greater control over the size and shape of the spatial objects, and it makes the output of queries more predictable. Working with a single real data set, as in SEQUOIA, ensures that the data is representative of some real world situation, but provides less opportunity for testing software in different scenarios. However, in VESPA, of the many parameters that could potentially be varied in data generation programs, this paper only varies the scale of the data sets generated. Further work on automatic generation of spatial data sets is described in [11, 22].

The benchmark is performed on collections of geometric objects representing features on a map. The resulting data sets contain simpler and more regular data than real maps, but are easy to generate and are usable to test a wide range of different cases. The benchmark uses data sets that represent land ownership, states, land use, roads, streams, gas lines and points of interest.

The land ownership data set forms a base for the generated data. Land ownership is represented by an  $n$  by  $n$  grid of hexagons. The size of all data sets is given relative to the number  $n$  of hexagons used to represent land ownership.

**Landown** The data set *landown* in Table 1 consists of  $n^2$  hexagons arranged in  $n$  rows with  $n$  columns. Cardinality:  $n^2$

Field Name	Field Definition	Comments
land_nr	integer	external ID
land	polygon	regular hexagon
owner	string[15]	n different names
value	float	generated randomly

**Table 1.** Data set landown

Field Name	Field Definition	Comments
statename	string[15]	generated randomly
state	polygon	regular hexagon
population	integer	random number < 10,000,000

**Table 2.** Data set states

**States** The data set *states* in Table 2 contains hexagons, with  $n/3$  rows consisting of  $n/3$  hexagons. Cardinality:  $(n/3)^2$

**Landuse** The data set *landuse* in Table 3 consists of triangles, with six triangles placed into each land ownership hexagon. Cardinality:  $6 * n^2$

Field Name	Field Definition	Comments
landusename	string[15]	generated randomly
land	polygon	regular triangle
landusetype	string[15]	1 from 5 landuse types enumerated cyclically

**Table 3.** Data set landuse

**Roads** The data set *roads* in Table 4 contains lines describing a network. It is built of slanting lines with  $n/2$  broadly horizontal and  $n/2$  broadly vertical directions. At each intersection point a new line segment starts. Therefore, a road consists of  $n/2 + 1$  line segments. Cardinality:  $2(n/2)$

Field Name	Field Definition	Comments
roadname	string[15]	randomly generated
road	line	line consisting of $n/2$ segments
roadtype	string[15]	three roadtypes enumerated cyclically
lanes	int	iterates from 1 to 6
width	float	computed from lanes

**Table 4.** Data set roads

**Streams** The data set *streams* in Table 5 consists of lines representing a binary tree. The root of the binary is located in the lower left corner. The depth of the tree is  $\sqrt{n}$ . Cardinality:  $2^{\sqrt{n}+1} - 2$

**Gas Lines** The data set *gaslines* in Table 6 also contains lines presenting a tree. Its depth is  $\sqrt{n} + 1$  and therefore it contains about twice as many items as

Field Name	Field Definition	Comments
streamname	string[15]	randomly generated
stream	line	1 segment connecting two points
width	float	

**Table 5.** Data set streams

the data set streams. The root of this tree is located in the lower right corner of the map. Cardinality:  $2^{\sqrt{n+2}} - 2$

Field Name	Field Definition	Comments
gasline_ID	integer	external ID
gasline	line	1 segment connecting two points

**Table 6.** Data set gasline

**Points of Interest** The data set *POI* (points of interest) in Table 7 contains points which are placed on  $n$  sloping, parallel lines with consistent distances. Cardinality:  $n^2$

Field Name	Field Definition	Comments
POIname	string[15]	randomly generated
POI	point	
POItype	string[15]	6 types enumerated cyclic

**Table 7.** Data set POI (points of interest)

**Scaling The Benchmark** The scaling of the benchmark is done by two parameters: the size of the map and the size of each data set. Although the data generator can generate data sets of any size, three data sets are used later in the paper with PostgreSQL and ROCK & ROLL, referred to in Table 8 as small, medium and large. For polygons and lines, this table gives both the number of features and the number of segments used to build the features.

Most of the data sets grow by about a factor of 10 with each size step. Exceptions are the data sets gaslines and streams, since they are represented as binary trees. For the data set roads, the number of roads grows about a factor of 3.3, while the number of line segments grows by a factor of 100.

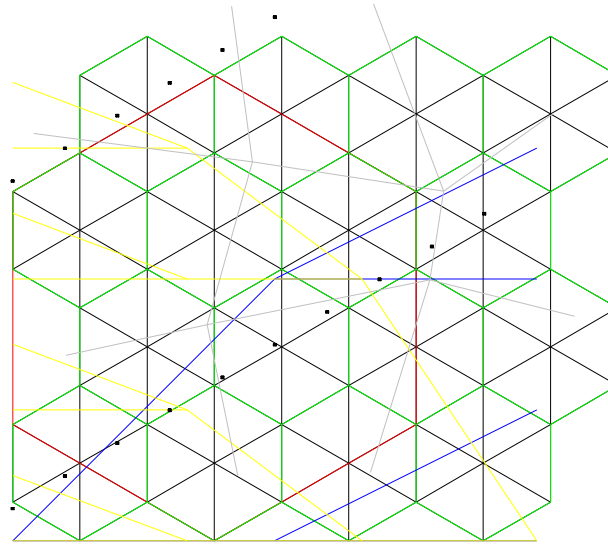
**Data Generation** The data is generated by a C++ program which provides classes for all the data sets used. The program has the command line gen <Scale

Feature	small	medium	large
scaling factor $n$	10	33	100
mapsize	1000	10,000	100,000
states polygons	9 (54)	121 (726)	1089 (6534)
landown polygons	100 (600)	1,089 (6,534)	10,000 (60,000)
landuse polygons	600 (1800)	6,534 (19602)	60,000 (180,000)
streams lines	14 (14)	62 (62)	2,046 (2046)
gaslines lines	30 (30)	126 (126)	4,096 (4,096)
roads lines	10 (60)	32(544)	100 (5,100)
POI points	100	1089	10,000

**Table 8.** Cardinalities. Values in brackets show the number of segments.

**Factor** <mapsize>, where <Scale Factor> is the number  $n$  used to describe the cardinalities, and <mapsize> specifies the size of the map. For each data set an ASCII file is written, which contains the list of coordinates for each feature.

A sample of the generated spatial data is provided in figure 1.



**Fig. 1.** Drawing of spatial parts of benchmark features

## 2.2 Tasks

This section provides an informal description of the tasks carried out as part of the benchmark. These tasks include updates to the database, construction of derived spatial values and queries.

## Updates

**Task 1: Insert new objects:** This involves inserting new information into the database, updating all relevant indexes and topological information stored.

- add a row of hexagons to the data set landown.
- add the centre points of the hexagons from the data set states to POI.

**Task 2: Delete objects:** This involves removing the spatial and aspatial data from the database that was inserted by Task 1.

**input:** landown, poi

**output:** modified data sets landown, poi

Cardinality:  $n + n$

## Set Operations

**Task 3: Union polygons:** Union all landuse polygons with neighbours of same landuse type.

**input:** landuse

**output:** new data set containing the result

Cardinality:  $\approx 0.8 * |landuse|$

## Containment Operations

**Task 4: Polygon contains point:** Retrieve all POI which are inside Scotland

**input:** states, POI

**output:** POI.poiname, POI.poitype

Cardinality:  $\approx 9 = \frac{|POI|}{|states|} = \frac{n^2}{(\frac{n}{3})^2}$

**Task 5: Polygon contains line:** Retrieve all streams which are completely inside Scotland.

**input:** streams, states

**output:** streams.streamname

Cardinality:  $\approx 3$

**Task 6: Polygon contains polygon:** Retrieve all forests which are completely inside Scotland.

**input:** landuse, states

**output:** landuse.landusename

Cardinality:  $\approx 11 \approx \frac{|landuse|/|landusetype|}{|states|} = \frac{6n^2/5}{(\frac{n}{3})^2}$

**Task 7: Line contains Line:** Retrieve all gas lines which are completely below a stream (with same location, since map is two dimensional).

**input:** gaslines, streams

**output:** gaslines.gasline\_ID

Cardinality:  $\leq \log_2 |gaslines|$ ,

**Task 8: Line contains point:** Retrieve all points of interest on the road called A1.

**input:** roads, POI

**output:** POI.poiname, POI.POIttype

Cardinality:  $\approx 0$

## Overlap Operations

**Task 9: Polygon overlap line:** Retrieve all roads which are at least partly inside Scotland.

**input:** states, roads

**output:** roads.roadname, roads.roadtype

Cardinality:  $\approx 4$

**Task 10: Polygon overlap polygon:** Retrieve all forests which are at least partly inside Scotland.

**input:** states, landuse

**output:** landuse.landusename

Cardinality:  $\approx 11 \approx \frac{|landuse|/|landusetype|}{|states|} = \frac{6n^2/5}{(\frac{4}{3})^2}$

**Task 11: Line overlap line:** Retrieve all gas lines which are at least partly below a stream.

**input:** gaslines, stream

**output:** gaslines.gasline\_ID

Cardinality:  $\leq \log_2|gaslines|$ , as in Task 10

## Intersect Operations

**Task 12: Line intersect line:** Retrieve all streams crossed by a road

**input:** streams, roads

**output:** streams.streamname

Cardinality:  $\leq |streams|$

**Task 13: Line intersect polygon:** Retrieve all roads which go through a forest.

**input:** landuse, roads

**output:** roads.roadname

Cardinality:  $\leq |roads|$

## Adjacent Operations

**Task 14: Polygon adjacent polygon:** Retrieve all states which border the state Scotland.

**input:** states

**output:** states.statename

Cardinality: 6

**Task 15: Polygon adjacent line:** Retrieve all polygons from landuse that are adjacent to the stream Dee.

**input:** landuse, streams

**output:** landuse.landusename

Cardinality:  $\approx 0$

**Task 16: Line connection:** Retrieve all streams connected directly to the Dee.

**input:** streams

**output:** streams.streamname

Cardinality: 4

### Search inside area

**Task 17: Buffer search:** Retrieve all polygons from landuse which are less than 200m from a stream.

**input:** landuse, streams

**output:** landuse.landusename, landuse.landusetype

Cardinality:  $\leq |landuse|$

**Task 18: Window search:** Retrieve all the POI, which are inside a rectangle of 10 km  $\times$  10 km with the tourist information in the center of the rectangle.

**input:** POI

**output:** POI.poiname

Cardinality:  $\approx 1$

### Measurement Operations

**Task 19: Size of polygon:** Compute the size of all polygons in landuse.

**input:** landuse

**output:** landuse.landusename, area(landuse.land)

Cardinality:  $|landuse|$

**Task 20: Length of line:** Compute the length of all streams.

**input:** streams

**output:** streams.streamname, length(streams.streamname)

Cardinality:  $|streams|$

**Task 21: Distance to constant:** Compute for all POI the distance to a specific tourist information office.

**input:** POI

**output:** POI.poiname, distance(Constant, POI.POI)

Cardinality:  $|POI|$

### Analysis Operations

**Task 22: Spatial aggregation:** Compute the size of the area overlapped by the polygons from landuse and states.

**input:** landuse, states

**output:** area( landuse  $\cup$  landown )

Cardinality: 1

**Task 23: Transitive closure:** Find all the streams that might be affected by salmon nets at the mouth of the river Dee.

**input:** streams

**output:** streams.streamname

Cardinality:  $4|streams|$

## Aspatial operations

**Task 24: Aspatial selection:** Retrieve all roads of type motorway with at least 3 lanes.

**input:** roads

**output:** roads.roadname, roads.lanes

Cardinality:  $\frac{4}{6}|roads|$

**Task 25: Aspatial join:** Select all stream and roads that have the same width.

**input:** streams, roads

**output:** streams.streamname, roads.roadname, streams.width

Cardinality:  $\frac{n}{3} \frac{2\sqrt{n+1}-2}{3}$

**Task 26: Aspatial aggregation:** Compute the sum of values of land for each owner.

**input:** landown

**output:** landown.owner,  $\Sigma$ landown.value

Cardinality:  $\sqrt{|landown|}$

## 3 The Benchmark in PostgreSQL

PostgreSQL is an object-relational database developed at the University of California at Berkeley [24] <sup>1</sup>. It is a derivative of the influential POSTGRES system [21], which pioneered work on the extension of the relational model with abstract data types and active rules. In this context, it is the abstract data type (ADT) facility that is of most interest, in that it has been used to extend the primitive types of PostgreSQL with vector spatial data types. The ADTs supported by PostgreSQL are not unlike those that are beginning to be supported by relational database vendors.

The ADTs in PostgreSQL allow programmers to introduce new types into the kernel of the database by:

1. Defining *input* and *output* functions that map from an external (string) representation of an instance of the type to the internal, stored form of the instance. For example, a point that is represented to the user as the string  $(1.0, 2.0)$  could be represented internally as a pair of reals stored in an R-tree.
2. Defining a collection of functions that take the user defined types as arguments and/or return them as results. This would allow, for example, a *polygon\_area* function to take as input a value of type *polygon* and to return a real.

The functions that define new types are written in C, and are dynamically linked into the PostgreSQL system at runtime. There is also an interface that allows the indexing facilities of the basic system to be extended to support indexing of ADTs, for example, to allow spatial indexes to be used with spatial data types. The ADTs of PostgreSQL can be considered to provide low-level and

---

<sup>1</sup> The URL of the current release of PostgreSQL is <http://www.postgresql.org>

potentially dangerous facilities for adding efficient extensions to the database for particular domains. They are potentially dangerous because incorrect ADT definitions can easily corrupt a database. However, they help to overcome the limited programming facilities that have traditionally restricted the use of the relational model for storing complex data types, such as those associated with spatial databases.

### 3.1 Data Model

The implementation of the benchmark exploited extensions to a collection of spatial data types that are supplied with the PostgreSQL system. The spatial types introduced into PostgreSQL are: *point*, which consists of a pair of real numbers; *lseg*, which consists of a pair of points; *path*, which consists of a list of points; and *polygon*, which consists of a list of points. All spatial types are indexed using an R-tree, and algorithms are largely derived from those provided in [17]. The PostgreSQL tables used by the benchmark are defined in figure 2.

```
CREATE TABLE landown (
    land_nr    int4,
    land       polygon,
    owner      char16,
    value      float8);

CREATE TABLE streams (
    streamname char16,
    stream     lseg,
    width      float8);

CREATE TABLE states (
    statename  char16,
    state      polygon,
    population int4);

CREATE TABLE gaslines (
    gasline_ID int4,
    gasline    lseg);

CREATE TABLE landuse (
    landusename char16,
    land         polygon,
    landusetype char16);

CREATE TABLE pois (
    poiname    char16,
    poi        point,
    poitype    char16);

CREATE TABLE roads (
    roadname    char16,
    road        path,
    roadtype    char16,
    lanes       int4,
    width       float8);
```

**Fig. 2.** Table definitions for benchmark in PostgreSQL

### 3.2 Tasks

All interaction with the database in the benchmark is carried out using SQL, which was always used embedded in C programs. For the sake of conciseness, only a few representative tasks are presented here, and only the SQL and not the surrounding C programs, are presented.

**Task 4: Polygon contains point:** This is a spatial join operation, where points of interest are retrieved when they are inside the polygon that is the `state` attribute of `Scotland`.

```
SELECT pois.poiname, pois.poitype
FROM pois, states
WHERE states.statename = "Scotland"
AND pt_in_poly(pois.poi, states.state)
```

**Task 9: Polygon overlap line:** This is similar to Task 7, except that the spatial join is based on the overlap operation involving paths and polygons.

```
SELECT roads.roadname, roads.roadtype
FROM roads, states
WHERE states.statename = "Scotland"
AND path_ovr_poly(roads.road, states.state)
```

**Task 14: Polygon adjacent polygon:** This is a topological self join using the adjacency relationship on states.

```
SELECT S1.statename
FROM states S1, states S2
WHERE S2.statename = "Scotland"
AND poly_adj_poly(S2.state, S1.state)
```

## 4 The Benchmark in ROCK & ROLL

The ROCK & ROLL deductive object-oriented database is built around three components:

**The object model OM:** The underlying data model is an object-oriented semantic data model that supports a conventional set of object modelling concepts – object identity, inheritance, sets, lists, aggregates, etc.

**The imperative language ROCK:** The language ROCK allows OM objects to be created, retrieved and manipulated. It provides a conventional set of programming constructs (loops, if statements, etc), and can be used to define methods on OM types.

**The deductive language ROLL:** The language ROLL allows deductive rules to be defined that express derived properties of the objects stored in the database in a declarative manner. ROLL can be used to express both queries and methods, but contains no facilities for describing control flow or updates. This latter feature makes ROLL amenable to optimisation using extensions of earlier deductive query optimisation techniques [7].

The fact that ROCK and ROLL share the same type system, namely OM, means that they can be integrated without manifesting the impedance mismatches that are characteristic of other multi-language systems [3].

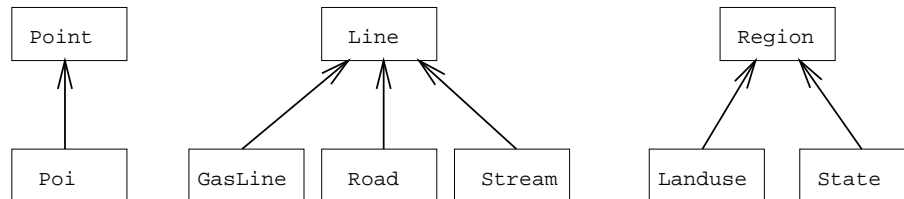
In the ODESSA (Object-oriented DEductive Spatial Systems Architecture) project [9], the kernel of ROCK & ROLL has been extended to provide support for the ROSE algebra [13]. The ROSE algebra is a vector spatial algebra that maps all spatial values onto a finite resolution grid called a realm. The operations of the ROSE algebra have desirable computational complexity[12], and ODESSA exploits a novel implementation strategy whereby the realm is not explicitly stored [16]. The implementation of ODESSA also corrects aspects of the original ROSE algebra proposal, in which problems arose with certain geometries.

The ROSE algebra provides three spatial data types: **points** – a set of point values; **lines** – a set of paths connecting realm points; **regions** – a set of areas, possibly containing holes. All operations on the ROSE algebra are closed – they return values that are themselves either scalar or points, lines or regions values.

The benchmarked implementation of ROCK & ROLL uses the commercial object database ObjectStore for persistence.

#### 4.1 Data Model

In ODESSA, **points**, **lines** and **regions** have been added to the OM data model as primitive types. This means that OM objects can have attributes that take on spatial values. The benchmark builds upon the straightforward class hierarchy given in figure 3.



**Fig. 3.** Inheritance hierarchy for benchmark

The OM classes **Point**, **Line** and **Region** each have an attribute of type **points**, **lines** and **regions** respectively. For example, the definition of **Region** is illustrated in figure 4. These classes are used as placeholders for generic functionality, such as that used to populate the database from text files. These classes are then subclassed by the domain classes used by the benchmark. For example, **State** is defined as in figure 5.

#### 4.2 Queries

All interaction with the database takes place using ROCK & ROLL programs. In practice, any task that can be carried out using ROLL can also be carried

```

type GEO.Region:
  properties:
    public:
      RegionsRep : regions;
  ROCK:
    new();
    readData();
    print();
    ...
end_type;

```

**Fig. 4.** Type definition for region

```

type GEO.State:
  specialises: Region;
  properties:
    public:
      Name: string,
      Number: int;
  ROCK:
    new();
    readData();
    print();
end_type

```

**Fig. 5.** Type definition for State

out using ROCK, but ROLL often allows more concise query expression and is declarative – an optimiser seeks to identify an efficient way of evaluating ROLL queries and rules. The following code fragments illustrate how a range of benchmark tasks are carried out using ROCK & ROLL.

**Task 4: Polygon contains point:** This is a spatial join operation, where points of interest are retrieved if they are inside a region. This can be expressed in either ROCK or ROLL. The first version illustrated below uses ROLL to obtain in *s* the object identifier of the State with name equal to Scotland, and then uses a ROCK foreach loop to iterate through the points of interest, checking to see if they are inside *s*.

```

var s:= [ any S | get_Name@S:State == "Scotland" ];

foreach p in Poi do begin
  if (get_PointsRep@p inside get_RegionsRep@s)
    write get_Name@p, nl;
end

```

The following version relies more on the embedded ROLL query, and the way in which the query is evaluated is implicit; an explicit ROCK loop is used only to generate the results:

```
var ps := [ all P | get_Name@S:State == "Scotland",
                get_RegionsRep@S == SR,
                get_PointsRep@P == PR,
                PR inside SR ];

foreach p in ps do
begin
    write "Point ", get_Name@p, " lies inside Scotland", nl;
end
```

**Task 9: Polygon overlap line:** The following ROLL query retrieves all Roads that either intersect with Scotland or that are inside Scotland.

```
[ all R | get_Name@S:State == "Scotland",
            get_RegionsRep@S == SR,
            get_LinesRep@R == RR,
            RR intersects SR
        ;
    get_Name@S:State == "Scotland",
    get_RegionsRep@S == SR,
    get_LinesRep@R == RR,
    RR inside SR ];
```

**Task 14: Polygon adjacent polygon:** Self joins are straightforward to express in logic languages, as can be seen from the following:

```
[ all A | get_RegionsRep@A:State == AR,
            get_RegionsRep@S:State == SR,
            AR adjacent SR,
            get_Name@S == "Scotland" ];
```

## 5 Results

This section presents the results of running the benchmark on the PostgreSQL and ODESSA platforms, with a view to allowing comparison of the different implementations and scaling of the specific approaches. The figures were obtained on a 143 Mhz Sun UltraSPARC1 with 64Mbytes of RAM. All figures are elapsed times, and are the average of three distinct runs.

Tables 9, 10 and 11 give the times obtained for PostgreSQL, ROCK and ROLL, respectively, so that the scalability of the individual approaches can be assessed. The following can be observed:

Task	Small	Medium	Large
1	2.50	3.20	4.20
2	1.50	5.50	55.90
3	33.20	3803.50	20419.20
4	0.95	1.33	4.30
5	0.90	0.95	1.85
6	1.13	3.08	20.95
7	1.00	3.00	2176.03
8	0.93	1.33	6.67
9	0.97	1.25	3.67
10	1.10	2.75	17.77
11	1.30	3.38	2118.10
12	1.30	3.50	622.63
13	1.75	13.97	317.50
14	0.90	1.25	3.67
15	1.10	3.50	24.98
16	0.90	0.98	1.90
18	0.90	1.38	5.23
19	1.45	6.70	98.71
20	0.90	1.08	2.80
23	1.00	1.90	4.50
24	0.90	0.95	0.98
25	0.90	1.50	63.77
26	1.00	1.50	5.40

**Table 9.** Benchmark results for PostgreSQL for all data sets.

Task	Small	Medium	Large
1	15.83	43.50	130.57
2	4.89	60.65	637.19
3	3.09	129.27	806.86
4	0.74	6.54	113.15
5	0.03	0.08	8.66
6	2.52	95.89	163.15
7	0.05	1.05	944.60
8	0.08	2.86	144.12
9	0.10	10.47	23.66
10	8.15	80.61	195.18
11	0.21	1.22	1077.11
12	0.48	52.15	4142.08
13	0.22	103.12	1252.53
14	0.01	0.07	7.12
15	0.10	42.29	209.14
16	0.01	1.53	22.96
19	4.53	128.84	1522.61
20	0.00	0.01	0.04
23	0.41	4.59	37.98
24	0.00	0.15	11.56
25	0.01	0.12	14.20
26	1.09	51.36	1261.26

**Table 10.** Benchmark results for ROCK for all data sets.

Task	Small	Medium	Large
1	6.68	85.55	1769.32
2	20.99	785.20	4135.72
4	1.10	7.22	232.88
5	0.06	0.08	16.30
6	2.78	148.55	669.63
7	0.01	1.15	972.16
8	0.10	4.86	284.31
9	0.14	4.99	30.84
10	0.17	307.35	1098.68
11	0.39	1.12	1195.64
12	0.57	69.87	4365.22
13	5.23	96.48	570.53
14	0.01	0.02	4.30
15	0.09	124.97	589.70
16	0.01	2.56	27.92
19	2.40	89.28	1055.25
20	0.01	0.30	7.73
23	1.50	48.20	255.72
24	0.00	0.16	12.76
25	0.01	0.41	18.57
26	0.94	44.25	1139.89

**Table 11.** Benchmark results for ROLL for all data sets.

**PostgreSQL:** Most of the benchmark tasks are supported rather well by PostgreSQL. The current ADTs do not support buffer search (*Task 17*). The transitive closure query (*Task 23*) has been carried out using the recursive facilities of PostgreSQL.

The performance of PostgreSQL is generally good, and most tasks scale well across the different data sets. This stems from the fact that PostgreSQL is able to make good use of its spatial index during query evaluation to restrict the number of pairwise comparisons of spatial values carried out. Where PostgreSQL scales less well in query tasks, for example in Tasks 7, 11 and 12, this is probably because the minimum bounding rectangles around streams are often quite large, and thus are not very effective at narrowing the search space.

**ROCK:** Most of the benchmark tasks have been supported in ROCK. Those that are absent include buffer search (*Task 17*) and measurement operations (*Tasks 21,22*), as these are problematic in the ROSE algebra due to the distortions associated with the finite resolution geometry.

The implementations of most tasks in ROCK involve explicit loops, sometimes nested, through stored collections, testing for spatial properties. This means, broadly speaking, that the best ROCK can hope to do is scale in proportion with the number of times that the most nested statement is executed. For example, in *Task 4*, the containment test is executed approximately 100 times in the small data set, 1000 times in the medium data set and 10000 times in the large data set. The increase in task time for small to medium of 0.74 to 6.54 seconds is broadly in line with the number of loop iterations, while the increase for the large data set to 113.15 seconds is slightly slower than might have been hoped for. The implementation of the ROSE algebra, in which intersections are computed and stored explicitly at insertion time, means that the more intersections exist in a data item, the slower will be the algorithms that run over the data item. Changes to the number of segments that intersect with the boundary of Scotland could have the effect of slowing the intersection test, but this should not explain the jump in execution time on its own, as initial intersection tests involve minimum bounding rectangles. Where the ROCK times deteriorate rapidly with growing data sets, this is where the number of inner loop executions is also large. For example, in *Task 11*, the number of iterations in the small, medium and large data sets are, respectively, (14×30) 420, (62×126) 7812 and (2046×4096) 8,280,416, so the time taken to perform individual membership tests shows no sign of increasing as the data set grows.

**ROLL:** To keep its logical semantics straightforward, ROLL is not able to update the underlying database (as in *Task 1*) or to construct new spatial values (as in *Task 3*). Thus where figures are included for ROLL for such activities, ROLL is used to carry out only part of the relevant task.

The implementations of the tasks in ROLL do not make explicit how the tasks should be carried out. In practice, query planning is carried out by the optimiser described in [7]. This optimiser has been extended so that it is aware of the spatial operators that can appear in queries, but currently does

no query planning that takes account of the semantics of spatial constructs. For example, it does not make use of the R-tree that is used to store ROSE algebra constructs (and which is used extensively when the database is updated). As a result, ROLL queries, like their ROCK counterparts, essentially execute nested loop joins over collections of spatially referenced objects, and scale broadly in line with the number of spatial operations in the inner loop. An extension to the ROLL optimiser could therefore have a significant effect on the time taken to evaluate most of the example queries.

Comparing ROLL with ROCK in the benchmark, ROLL queries are occasionally faster than their ROCK counterparts, but most perform similarly, or up to 5 times slower, for larger data sets. This is largely due to the fact that ROLL uses a set-oriented evaluation strategy, and thus allocates and deallocates significant amounts of store during query processing. The set-oriented semantics turn out not to be particularly effective in avoiding repetition of effort in this benchmark.

The following can be observed in terms of the relative merits of the different approaches:

1. Inserting new data into the database and deleting existing data from the database is generally quicker in PostgreSQL than in ODESSA. This is to be expected, as considerable effort is expended at insertion time in the ROSE algebra to resolve interactions between values that are being updated and other spatial values. This up-front effort in the ROSE algebra is central to the clean semantics that it supports, and allows other spatial operations to be supported efficiently [12].
2. The effective use of spatial indexes at query time is generally much more important than the performance of pairwise operations on spatial values. The fact that the ROSE algebra operations have attractive complexity measures compared with those implemented in PostgreSQL makes little difference in most of the query tasks, and in most cases PostgreSQL scales much better than ODESSA. The only task in which the complexity of the pairwise operations seems to dominate is *Task 3*, where ODESSA scales much better than PostgreSQL.
3. The poor performance of ROLL in the benchmark stems largely from the failure of the optimiser to exploit the R-tree that exists in the ODESSA system. This in turn is because the ROLL optimiser is largely a logical optimiser [7], and a key requirement here is for a range of physical optimisations.

## 6 Conclusions

This paper has presented a vector based spatial benchmark that can be used to test both the functionality and performance of spatial database systems. The benchmark includes a range of query and update tasks over synthetic data sets, and can be implemented with moderate effort on different platforms. The intention in providing a fairly large number of test queries and programs has been to

allow fine grained analysis of the performance of systems. This was important to us in the development of the ODESSA system, and we believe should be of interest to the developers of future spatial database systems.

The comparison presented in this paper, between PostgreSQL and ODESSA, shows that the benchmark can be used to differentiate between the performance of different spatial database systems, and that the tests stretch the different systems in different tasks. The benchmark can also be used as a test suite for evaluating the reliability of new database systems and implementation techniques. For example, it was implementing the VESPA benchmark on ODESSA that revealed to us an error in the underlying geometry on which the ROSE algebra builds; the resolution of this problem is described in [8].

**Acknowledgements:** This work is supported by the UK Engineering and Physical Sciences Research Council, whose support we are pleased to acknowledge. We are also grateful for the contribution of Reto Hafner, who implemented most of the benchmark in ROCK & ROLL.

## References

1. A.I. Abdelmoty, N.W. Paton, M.H. Williams, A.A.A. Fernandes, M.L. Barja, and A. Dinn. Geographic Data Handling in a Deductive Object-Oriented Database. In D. Karagiannis, editor, *Proc. 5th Int. Conf. on Databases and Expert Systems Applications (DEXA)*, pages 445–454. Springer-Verlag, 1994.
2. D. Arctur, E. Anwar, J. Alexander, S. Charkravarthy, Y. Chung, M. Cobb, and K. Shaw. Comparison and benchmarks for import of VPF geographic data from object-oriented and relational database files. In *Proc. SSD 95*, pages 368–384. Springer-Verlag, 1995.
3. M.L. Barja, N.W. Paton, A.A.A. Fernandes, M.H. Williams, and A. Dinn. An Effective Deductive Object-Oriented Database Through Language Integration. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. on Very Large Data Bases (VLDB)*, pages 463–474. Morgan-Kaufmann, 1994.
4. P.A. Boncz and M.L. Kersten. Monet: An Impressionist Sketch of an Advanced Database System. In *Proc. Basque Int. Wshp. on Information Technology*, pages 240–251. IEEE Press, 1995.
5. M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases*, CA 94303-9953, 1990. Morgan Kaufman Publishers, Inc.
6. S.W. Dietrich, M. Brown, E. Cortes-Rello, and S. Wunderlin. A Practitioners Introduction to Database Performance Benchmarks and Measurements. *The Computer Journal*, 35(4):322–331, 1992.
7. A. Dinn, N.W. Paton, M.H. Williams, A.A.A. Fernandes, and M.L. Barja. The Implementation of a Deductive Query Language Over an Object-Oriented Database. In T.W. Ling, A.O. Mendelzon, and L. Vieille, editors, *Proc. 4th Intl. Conf. on Deductive Object-Oriented Databases*, pages 143–160. Springer-Verlag, 1995.
8. A. Dinn, M.H. Williams, and N.W. Paton. Ensuring Geometric Consistency in the ROSE Algebra for Spatial Datatypes. In *submitted for publication*, 1997.
9. A.A.A. Fernandes, A. Dinn, N.W. Paton, M.H. Williams, and O. Liew. Extending a Deductive Object-Oriented Database System with Spatial Data Handling Facilities. *Information and Software Technology*, 41:483–497, 1999.

10. K. Gardels. SEQUOIA 2000 and Geographic Information: The GuerneWood Geoprocessor. In T. Waugh and R. Healey, editors, *Proc. SDH*, pages 1072–1085. Taylor & Francis, 1994.
11. O. Gunther, V. Oria, P. Picouet, J-M Saglio, and M. Scholl. Benchmarking Spatial Joins A La Carte. In *Proc. SSDBM*, pages 32–40. IEEE Press, 1998.
12. R.H. Guting, T. de Ridder, and M. Schneider. Implementation of the ROSE Algebra: Efficient Algorithms for Realm-Based Spatial Data Types. In M.J. Egenhofer and J.R. Herring, editors, *Proc. 4th Int. Symposium on Large Spatial Databases (SSD)*, pages 196–215. Springer-Verlag, 1995.
13. R.H. Guting and M. Schneider. Realm-Based Spatial Data Types: The ROSE Algebra. *VLDB Journal*, 4(2):243–286, 1995.
14. E.G. Hoel and H. Samet. Benchmarking Spatial Join Operations with Spatial Output. In *Proc. VLDB*, pages 606–618, 1995.
15. S. Morehouse. ARC/INFO: A Geo-Relational Model for Spatial Information. In *Proceedings of 7th Int. Symposium on Computer Assisted Cartography*, pages 388–398, Washington, DC, 1986.
16. V. Muller, N.W. Paton, A.A.A. Fernandes, A. Dinn, and M.H. Williams. Virtual Realms: An Efficient Implementation Strategy for Finite Resolution Spatial Data Types. In M.J. Kraak and M. Molenaar, editors, *Proc. 7th SDH*, pages 697–709. Taylor & Francis, 1996.
17. J. O'Rourke, editor. *Computational Geometry in C*. Cambridge University Press, New York, 1994.
18. J. Patel. Building a Scalable Geo-Spatial DBMS: Technology, Implementation, and Evaluation. In *Proc. SIGMOD Conf.*, pages 336–374. ACM Press, 1997.
19. M. Stonebraker, R. Agrawal, U. Dayal, E. Neuhold, and A. Rueter. DBMS Research At A Crossroads: The Vienna Update. In *Proc. of the 19th VLDB*, pages 688–692, Dublin, Ireland, 1993. R. Agrawal et al (Eds).
20. M. Stonebraker, J. Frew, K. Gardens, and J. Merideth. The sequoia 2000 storage benchmark. In *Proc. ACM SIGMOD*, pages 2–11, 1993.
21. M. Stonebraker and G. Kemnitz. The POSTGRES Next-generation Database Management System. *Communications of the ACM*, 34(10):78–92, October 1991.
22. T. Theodoridis, R. Silva, and M. Nascimento. On the Generation of Spatiotemporal Datasets. In *Proc. 6th Int. Symposium on Spatial Databases*, pages 147–164. Springer Verlag, 1999.
23. Y. Theodoridis, T. Sellis, A.N. Papadopoulos, and Y. Manolopoulos. Specifications for Efficient Indexing in Spatiotemporal Databases. In *Proc. SSDBM*, pages 123–132. IEEE Press, 1998.
24. A. Yu and J. Chen. The Postgres95 User Manual. Technical report, Computer Science Division, Dept of EECS, University of California at Berkeley, 1995.