[11] C. Esperanca and H. Samet. Representing orthogonal multidimensional objects by vertex lists. In C. Arcelli, L. P. Cordella, and G. Sanniti di Baja, editors, *Aspects of Visual Form Processing*, pages 209--220, Singapore, 1994. World Scientific.

[12] G. Graefe. Volcano--an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120--135, February 1994.

[13] R. H. Güting. Gral: An extensible relational system for geometric applications. In *Proceedings of the 15th International Conference on Very Large Databases*, pages 33--44, Amsterdam, August 1989.

[14] G. Hjaltason and H. Samet. Ranking in spatial databases. In M. J. Egenhofer and J. R. Herring, editors, *Proceedings of the Fourth Symposium on Spatial Databases*, Portland, ME, August 1995.

[15] B. C. Ooi, R. Sacks-Davis, and K. J. McDonell. Extending a DBMS for geographic applications. In *Proceedings of the Fifth IEEE International Conference on Data Engineering*, pages 590--597, Los Angeles, February 1989.

[16] J. A. Orenstein and F. A. Manola. Spatial data modeling and query processing in PROBE. Technical Report CCA-86-05, Computer Corporation of America, Cambridge, MA, October 1986.

[17] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, April 1994.

[18] N. Roussopoulos, C. Faloutsos, and T. Sellis. An efficient pictorial database system for PSQL. *IEEE Transactions on Software Engineering*, 14(5):639--650, May 1988.

[19] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[20] H. Samet and A. Soffer. Integrating images into a relational database system. Technical Report CS-TR-3371, University of Maryland, College Park, MD, October 1994.

[21] M. Stonebraker. Inclusion of new types in relational data base systems. In *Proceedings of the 2nd International Conference on Data Engineering*, pages 262--269, Los Angeles, CA, February 1986.

[22] Economics U.S Department of Commerce and Bureau of the Census Statistics Administration. Tiger/line census files, 1992. Technical documentation, 1993.

[23] A. Wolf. The DASDBS GEO-Kernel: Concepts, experiences, and the second step. In A. Buchmann, O. Günther, T. R. Smith, and Y. F. Wang, editors, *Design and Implementation of Large Spatial Databases, Proceedings of the First Symposium SSD'89*, pages 67—88. Springer-Verlag, Berlin, 1990. (also Lecture Notes in Computer Science 409).

5. Final Considerations

SAND is an on-going project. Most, but not all of the capabilities shown in this paper are implemented. The plan generation and optimization used in the library is still crude and needs to be enhanced. In particular, the heuristics used by the optimizer tend to favor plans using indices whenever possible, which not always results in an optimal plans. Cost estimation is currently not used at all.

In order to improve SAND's plan-generating strategies, we are presently considering a rule-based optimizer in the molds of GRAL [5] and dynamic query optimization as proposed for the Volcano system[8].

In its current state, however, SAND has already been proved of value. For instance, we have built the "Map Browser", an application that uses a simple graphical user interface to answer simple queries. All examples shown in this paper were produced with the aid of the Map Browser. Additionally, SAND has been used for the prototype of an image database [20].

References

[1] D. J. Abel. SIRO-DBMS: A database tool-kit for geographical information systems. *International Journal of Geographical Information Systems*, 3(2):103--116, April--June 1989.

[2] W. G. Aref and H. Samet. An approach to information management in geographical applications. In *Proceedings of the Fourth International Symposium on Spatial Data Handling*, volume 2, pages 589--598, Zurich, Switzerland, July 1990.

[3] W. G. Aref and H. Samet. Extending a DBMS with spatial operations. In O. Günther and H. J. Schek, editors, *Advances in Spatial Databases - 2nd Symposium, SSD'91*, pages 299--318, Berlin, 1991. Springer-Verlag. (also Lecture Notes in Computer Science 525).

[4] W. G. Aref and H. Samet. The spatial filter revisited. In T. C. Waugh and R. G. Healey, editors, *Sixth International Symposium on Spatial Data Handling*, pages 190--208, Edinburgh, Scotland, September 1994. International Geographical Union Comission on Geographic Information Systems, Association for Geographical Information.

[5] L. Becker and R. H. Güting. Rule-based optimization and query processing in an extensible geometric database system. *ACM Transanctions on Software Engineering*, 17(2):247--303, June 1992.

[6] R. Berman, M. Stonebraker, and L. Rowe. Geo-quel: A system for the manipulation and display of geographical data. *Computer Graphics*, 11(2):186--191, 1977.

[7] E. F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377--387, June 1970.

[8] R. K. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *SIGMOD 94*, pages 150--160, Minneapolis, Minnesota, May 1994.

[9] D. Comer. The ubiquitous B--tree. *ACM Computing Surveys*, 11(2):121--137, June 1979.

[10] M. J. Egenhofer. Spatial sql: A query and presentation language. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):86--95, February 1994.

```
# Create tmp relation with all lines of Georgia Ave
sand create tmp line line
set table_handle [select_plan sspring name=="Georgia"]
set tmp_handle [sand open tmp]
while {[$table_handle status]} {
   $tmp_handle setfrom $table_handle
   $tmp_handle insert
   $table_handle next
}
$tmp_handle close
$table_handle close

# Scan PMR sspring.line block-wise
set block_handle [sand open sspring.line -blockwise]
while {[$block_handle status]} {
   set block [$block_handle get]
   set tmp_handle [select_plan tmp {[distance line $block]<=0.01}]
   while {[$tmp_handle status]} {
      set table_handle [select_plan sspring {[intersects line $block]}]
      while {[$table_handle.status]} {
         if [distance $tmp_handle.line $table_handle.line]<=0.01 {
            puts [$table_handle get]
         }
         $table_handle next
      }
      $table_handle close
   }
   $tmp_handle close
}
$block_handle close
```

A few notes about the above script:

- In command "`sand create tmp line line`" the first `line` refers to the name and the second to the type of the single attribute in table `tmp`'s schema.
- Command "`$tmp_handle setfrom $table_handle`" copies all attributes with same name from one tuple buffer to the other. In this case, only attribute `line` will be copied.
- Option `-blockwise`, when used for opening a spatially organized table, means that the returned table handle will scan blocks and not tuples. Thus, when the contents of the tuple buffer for the handle is accessed (as in "`$block_handle get`"), a value of type `rectangle` is returned to indicate the position and extent of the block.

This plan demonstrates how the blocks of the PMR-quadtree can be used as bounding boxes (step 3 in the strategy), and thus permit a fast elimination of tuples that cannot satisfy the query. If we analyze this plan more carefully, however, we will notice that the temporary result `tmp` is a simple table, i.e., it has no spatial structure. This forces each block retrieved from the PMR-quadtree to be compared exhaustively with all elements of `tmp`. A possibly better alternative is to use a temporary result which is spatially structured. The idea is to make `tmp` a PMR-quadtree as well. This will allow the usage of a quadtree-based spatial join algorithm (see, for instance, [4]), instead of the tuple-by-tuple nested loop join performed in steps 3 and 4 above. This is a powerful technique, because it prevents the loss of information concerning the spatial proximity of features as multiple steps of a plan are executed.

to the spatial access methods in use. For the type of PMR-quadtree implemented by SAND, for instance, Aref and Samet [4] propose a spatial join algorithm that requires access to the relative order and dimensions of each quadtree block appearing in each table. The SAND kernel addresses this problem by supporting a special kind of scan iterator for PMR-quadtree tables that implements block-by-block retrieval.

In order to illustrate this concept, consider a query where we want to list all streets within a distance of 1 mile from Georgia Ave. Figure 2 shows a map with the `line` attribute of all tuples that satisfy this query. A possible plan for this query could adopt the following strategy:

1. Perform a select plan to retrieve all tuples of `sspring` corresponding to Georgia Ave.
2. Create a temporary table `tmp` to store the line segments of the tuples retrieved by the select plan.
3. Scan the PMR-quadtree `sspring.line` block-by-block and select those blocks B that lie within one mile of a line segment L in `tmp`.
4. For each pair B/L, retrieve tuples of `sspring` whose `line` attribute is overlapped by B and test them against L.
5. Print those tuples that qualify.

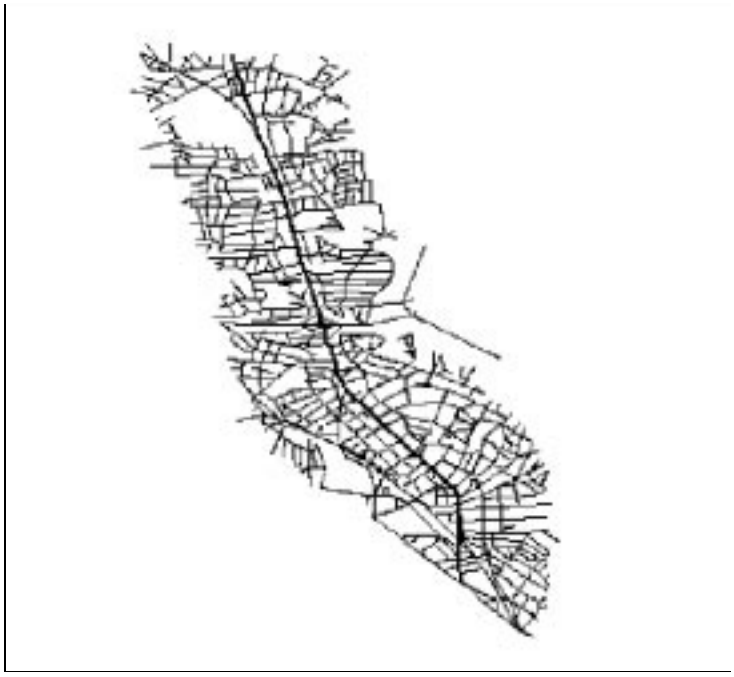

Figure 2: Example of spatial join. Map of Silver Spring showing line segments corresponding to Georgia Ave (thick line) and all line segments of streets within one mile of Georgia Ave (thin lines).

A SAND script for this strategy is shown below.

```
# Script to list all tuples of sspring  within one mile of any
# other tuple such that name="Georgia".
```

possible. Additionally, PMR-quadtree tables can be scanned in a number of spatially meaningful orders. Such scan orders are achieved by using special forms of the `first` command. For instance, it is possible to retrieve tuples in order of increasing or decreasing distance from a given feature by using a command called `firstcloseto`. Note that only one command is needed to initiate a scanning order, i.e., one particular form of `first`. In contrast, the `next` command performs the function of retrieving successive tuples independently of the scanning order.

The `firstcloseto` command implements incremental spatial ranking as described in [14]. The advantage of its incremental nature is that at any given moment of the scanning process, the algorithm has processed only enough quadtree blocks to determine the position of the current tuple within the scanning order.

It is important to realize the difference between *explicit* scanning orders such as those implemented by sorting a relation or using a B-tree and *implicit* scanning orders such as the one initiated by the `firstcloseto` command. In the first case, the order in which the tuples are retrieved is a consequence of the sequential nature of the table, and therefore is limited to a few variations, e.g., increasing or decreasing value of a B-tree key. In the second case, the ordering is not "pre-computed", that is, the ordering is a function of parameters that define geometric relationships which can be infinitely varied. This is one of the reasons why the processing of spatial data is intrinsically more challenging.

As an example, we will use a PMR-quadtree table defined as an index for attribute `line` of `sspring`, to rewrite the query that returns all tuples within a certain distance of a point. In the script below, this index table, called `sspring.line`, is scanned in such a way as to retrieve the line segments in increasing order of distance from the point of interest. For each qualifying tuple of the index, the corresponding tuple of `sspring` is retrieved by using the `indexfrom` command. The scanning is interrupted if the line segment retrieved from the index is farther than one mile from the point of interest.

```
# Script to list all tuples of sspring in the area within one mile
# (approx. 0.01 in map units) of point (-77.03, 39.0).
# Uses only the SAND kernel.
set index_handle [sand open sspring.line]
set table_handle [sand open sspring]
set point "point -77.03 39.0"
$index_handle firstcloseto $point
while {[$index_handle status] &&
       [distance $index_handle.line $point]<=0.01} {
   $table_handle indexfrom $index_handle
   puts [$table_handle get]
   $index_handle next
}
$index_handle close
$table_handle close
```

Notice that the above plan could actually have been the plan produced by the library in our previous implementation. This example illustrates how an application is frequently able to obtain the desired results even in the absence of a query plan generator. This characteristic is important for a research tool such as SAND.

Most selection queries based on spatial predicates can be answered efficiently by a examining a PMR-quadtree table using an appropriate scan order. A tuple-by-tuple scan, however, is too poor a mechanism for performing join queries based on spatial predicates -- the so-called *spatial joins*. Spatial join algorithms usually take advantage of the clustering properties particular

In this script, `.r` refers to a graphical output window, `draw_sand` is a SAND command used for drawing and `-style`is an option that modifies drawing parameters such as color or line thickness. The output of this script is shown in Figure 1.

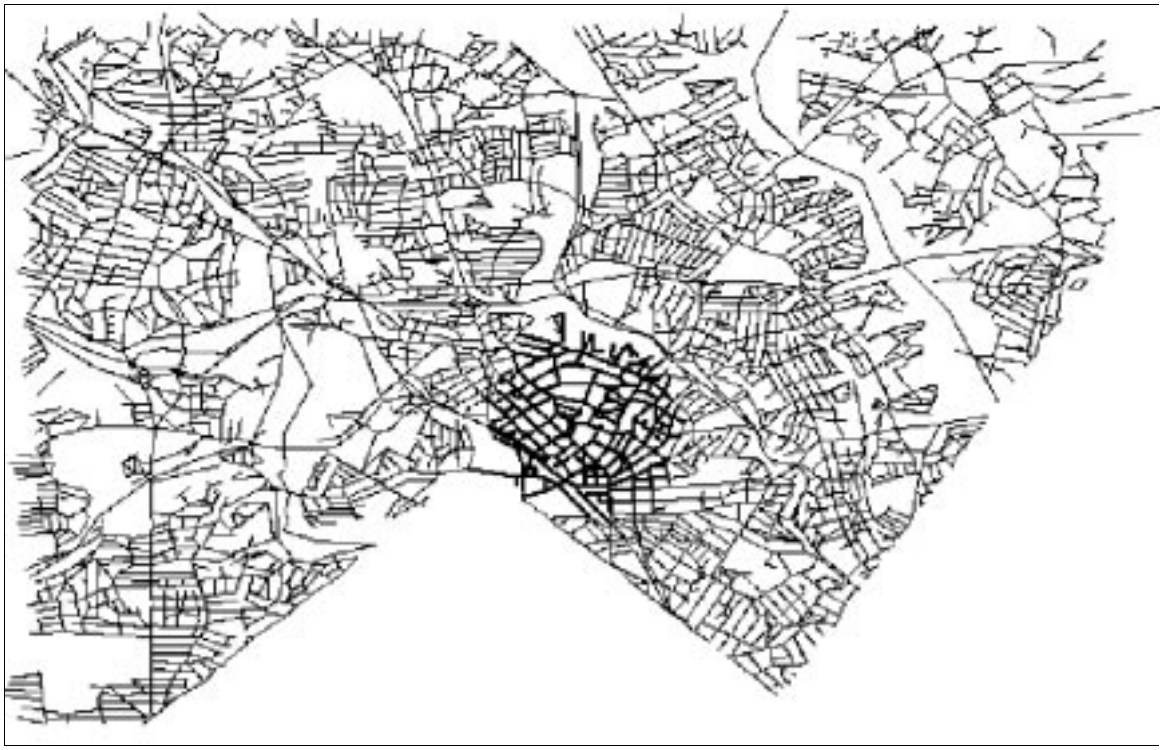

Figure 1: Map of a part of the city of Silver Spring, Maryland, obtained by drawing the value of attribute `line` of relation `sspring`. The highlighted streets (thicker lines) correspond to lines within 1 mile of point (-77.03, 39.0)

This example also illustrates how the application-oriented approach used in SAND can sometimes pay off in terms of a relatively simple implementation. In order to obtain the same results with a full-featured spatial database we would have to use a conventional programming language capable of accessing the query engine (say, embedded SQL). Another alternative would be to use a graphical presentation language like the one proposed by Egenhofer [10].

4.2. Spatially organized tables

In addition to being able to process spatial data at the attribute level, SAND supports two types of spatial access structures implemented as tables: PMR-quadtrees and region quadtrees [19].

A PMR-quadtree table organizes tuples spatially based on one of its attributes whose type must be one of SAND's spatial types, i.e., point, line, rectangle, polygon and region. As with any other table type, tuples in a PMR-quadtree table may be examined by applying methods `first` and `next`, in which case the tuples are retrieved in some arbitrary order. Such a scan order is used in situations where the only requirement is that all tuples be examined exactly once as fast as

Since the only primitive data type handled by Tcl is the character string, constants and variables representing spatial features must eventually be converted to and from strings of characters. Thus, for instance, string "point 1.0 2.0" represents a point in 2D with coordinates x and y equal to 1 and 2 respectively. This translation process, however, is relatively costly, and should be avoided in query evaluation plans. For this reason, the SAND kernel provides support for *attribute variables*. These are similar to regular Tcl variables, except that they need not be dereferenced explicitly (and thus converted to strings) when used in functions and predicates implemented by SAND. In particular, the tuple buffer associated with an open table handle can be accessed by means of attribute variables that are created automatically when the table is opened. Thus, for instance, after executing the command

```
set handle [sand open sspring]
```

five attribute variables are created, named `$handle.name`, `$handle.type`, `$handle.zipleft` etc.

As an example, consider a query plan for listing all all streets of `sspring` within 1 mile from a given point. One way to obtain such a plan would be to use the `select_plan`library function by specifying a suitable predicate:

```
# script to list all tuples of sspring in the area within one mile
# (approx. 0.01 in map units) of point (-77.03, 39.0)
set handle [select_plan sspring {[distance line "point -77.03,
39.0"]<=0.01}]
$handle first
while {[$handle status]} {
   puts [$handle get]
   $handle next
}
$handle close
```

Consider now a related query where we wish to obtain a *graphical* output of table `sspring` by drawing all line segments corresponding to attribute `line`, but *emphasizing* those line segments lying within one mile of point (-77.03, 39.0). Clearly, this query requires that all tuples of *sspring* be accessed, and thus it is very likely that a simple scan will prove to be an "optimal" plan. All that is required is to test each line segment for proximity with the given point in order to determine the proper drawing style. This is shown in the following script:

```
# script to draw all tuples of sspring, emphasizing
# those within one mile of point (-77.03, 39.0)
set handle [sand open sspring]
$handle first
while {[$handle status]} {
   if [distance $handle.line "point -77.03, 39.0"]<=0.01}] {
      .r draw_sand $handle.line -style 1
   } else {
      .r draw_sand $handle.line -style 2
   }
   $handle next
}
$handle close
```

The plan generating strategies used in the library vary in complexity from very crude to very sophisticated. Some, such as the ones described above, are modeled after relational algebra operators [7], while others are implementations of classic algorithms such as the index-scan or the hash-join algorithm. Since tables implemented both by the kernel and by the library share a common interface, query plans can be built by drawing on either. In other words, SAND offers table handling capabilities on various levels but does not impose a strict hierarchy among these levels. This arrangement permits applications to use high-level tools when these are available, and to resort to low-level tools either when higher level capabilities are unavailable or, sometimes, in an attempt to extract better performance. This open ended architecture is also valuable in the study of issues in spatial database optimization.

4. Spatial Data Handling

SAND supports spatial data at various levels. At the attribute level, five 2D geometric data types can be used in table schemas and can be processed with the aid of a set of functions and predicates. SAND also supports two types of spatially organized tables which can be accessed in several ways.

4.1. Spatial features

The following set of spatial attribute types (also referred to as *spatial features*) are currently implemented in SAND:

Attribute Type	Description
`point`	a point in 2D.
`polygon`	a simple (no holes, non-self-intersecting) polygon in 2D.
`rectangle`	an axes-aligned rectangle in 2D.
`region`	an arbitrary axes-aligned polygon in 2D.

The coordinate values of all spatial features are stored as double-precision floating point numbers. Polygons are stored as a list of its endpoints and regions are stored and manipulated with the aid of structures called *vertex lists* [11].

SAND also supports a series of spatial predicates and functions. Most of these are polymorphic, that is, they operate on any type of spatial features. For example:

`distance` *f1 f2* is a function that returns the Euclidean distance between spatial features *f1* and *f2*.

`intersects` *f1 f2* is a predicate that returns true if and only if features *f1* and *f2* occupy at least one common point in space.

`bbox` *f1 f2 ... fN* returns the smallest axes-aligned rectangle enclosing all features *f1 ... fN*.

3. The SAND library

The most important distinction between a script in SAND and a query expressed in a high-level query language such as SQL is that the latter does not usually describe a series of steps that are required to obtain the answer, whereas the former very often looks like a standard computer program. From the application point-of-view, the power of expression of a high-level query language is certainly advantageous, but not indispensable. A more important issue is how efficiently a query can be answered. In a standard database system, this issue is addressed by providing a query planner and optimizer. In SAND, this same issue is addressed by a library that is responsible for creating efficient plans.

As an example, consider a modification of the previous query so that the listing is restricted to streets corresponding to zipcode 20895. Such a query could be expressed in SQL by:

```
select * from sspring where zipleft=20895 or zipright=20895
```

One way to express this query in SAND is to modify the previous script so that the action of printing a tuple is conditioned to the satisfaction of the predicate. This approach, however, may prove to be inefficient. For instance, it may be the case that `sspring` possesses other access paths that allow the retrieval of tuples by zipcode directly. In a standard database system these factors are weighed by the query optimizer in order to provide the best possible query evaluation plan. In SAND, such a plan is created via a call to a library function. Some of the plan generating functions implemented by the SAND library are listed below:

`selection_plan` *name predicate* generates a plan to retrieve all tuples of table *name* which satisfy the given *predicate*.

`project_plan` *name attr1 attr2 ... attrN* generates a plan to retrieve from table *name* distinct tuples corresponding to the subset of attributes *attr1 attr2 ... attrN*.

`union_plan` *name1 name2* generates a plan to retrieve all distinct tuples from tables *name1* and *name2*.

`semi_join_plan` *name1 name2 predicate* generates a plan to retrieve all tuples of table *name1* such that there exists at least one tuple of *name2* which satisfies *predicate*.

The *predicate* argument used in some of these functions has the form of a common Tcl expression, except that attribute names are replaced by attribute values before its evaluation. The result of calling any of these functions is a table handle which can be used in the same way as the handles returned by the kernel `sand open` command. Thus, the above query may be answered in SAND by using the following script:

```
# script to list all tuples of sspring in the area of zipcode 20895
set handle [select_plan sspring zipleft==20895||zipright==20895]
$handle first
while {[$handle status]} {
   puts [$handle get]
   $handle next
}
```

For those not familiar with the syntax of Tcl, a few explanations are in order:

- The pound sign (#) used at the beginning of a line denotes a comment.
- The dollar sign ($) is used for variable dereferencing. Thus, construct "$*var*" denotes the value of variable *var*.
- `set` *var value* assigns *value* to variable *var*.
- The construct "[*string*]" used as a value means that *string* is to be evaluated as a command and the result used instead.
- `puts` is one of Tcl's file output commands.

One distinction that should be made at this point is that a table is not necessarily linked to a file or any kind of storage structure. It may perhaps be compared to the concept of *cursor* existing in many implementations of relational databases or, better still, to the concept of *iterator* used in the Volcano system [12]. For instance, the output of SAND's query plan generator is also a table, i.e., opening a query evaluation plan (QEP) is equivalent to starting its execution, with methods *get*, *first* and *next*serving the purpose of accessing the data computed by the plan. The SAND library is discussed in the next section.

The table abstraction serves as a common ground to the various access methods available in SAND. Again using object-oriented terminology, we say that the set of methods presented earlier define the simplest type of table, or a "base" table class. Other, more specialized tables, offer additional capabilities by means of additional methods; these constitute "derived" table classes. For instance, the SAND kernel presently supports direct access tables, buffer tables, B-tree tables [9], PMR-quadtree and region quadtree tables [19]. Each of these are supported by appropriate additional methods:

1. Direct access tables and buffer tables support a method called *goto* which, given a tuple identifier (i.e., a *tid*), loads the tuple buffer with the corresponding tuple.
2. B-tree and hash tables support the *find* method, which locates the tuple whose key is closest to (or equal to, in the case of hash tables) the one presently loaded in the tuple buffer, e.g. via the *set* method.
3. PMR-quadtrees offer methods related to the many varieties of spatial queries: window, closest to feature, incremental nearest, etc.
4. Region quadtrees offer methods for performing operations on raster maps, such as set-theoretic operations, buffer zones, connected component labeling, etc.

The creation and destruction of tables of any of the types described above is performed by commands `sand create` and `sand drop`, respectively. Also, since all of these table types are used for storage, their contents can be altered by the following two methods:

Method	Description
`insert`	the contents of the tuple buffer is added as a new tuple in the table
`delete`	the tuple most recently retrieved from the table is removed

Method	Description
`get`	returns the value of the tuple buffer
`set` *tuple value*	loads the tuple buffer with the given value
`first`	loads the tuple buffer with the first tuple in the table
`next`	loads the tuple buffer with the next tuple in the table
`status`	returns a boolean value that indicates whether the last call of an access method (e.g. `first`, `next`) was successful

To illustrate this mechanism, consider an example table called `sspring`. It is a relation containing geographical information about an area corresponding to the city of Silver Spring, Maryland. This table stores data provided by the Census Bureau [22] and has the following schema:

Attribute	Type	Description
`name`	char(30)	the name of a street
`type`	char(4)	street type (e.g., road, ave., lane)
`zipleft`	integer	zip code for the left side of the street
`zipright`	integer	zip code for the right side of the street
`line`	line	line segment corresponding to the geographic location of the street or a part thereof

If table `sspring` was part of a standard relational database, then in order to obtain a listing of its contents, we would use the query language supported by the database system (e.g., SQL) to form a statement like:

```
select * from sspring
```

Compare this to the following SAND script:

```
# script to list all tuples of sspring
set handle [sand open sspring]
$handle first
while {[$handle status]} {
   puts [$handle get]
   $handle next
}
$handle close
```

database systems did not provide enough flexibility to experiment with different data models, spatial indexing structures and query optimization techniques. In other words, we feel that the integration of spatial and non-spatial data processing is still an open problem that has to be attacked on a wide front and at many levels. We would like, however, to be able to use the result of efforts in that area from an application point-of-view, so that, as new capabilities are developed, applications that can take advantage of such capabilities can be created or modified with little effort. SAND was created in this spirit. It does not purport to be a full featured spatial database; instead, it is a programming environment dedicated to the development of applications dealing with spatial and non-spatial data.

SAND can also be viewed as a toolkit that is accessed primarily by means of an interpreted language. Thus, all capabilities offered by SAND take the form of commands in that language, and queries are usually expressed by means of short code fragments called scripts. This arrangement allows SAND to emulate different paradigms for query processing. Currently, for instance, SAND offers in its library a set of functions that implement operators of the relational algebra[7]. No query language, however, has been devised for SAND, even though some high-level query languages have been proposed in the past which would be adequate to express the class of queries that SAND proposes to answer [6, 10, 15, 18].

The rest of this paper is organized as follows. Section 2 discusses the overall architecture of the SAND kernel. Section 3 presents query processing with the aid of the SAND library. Section 4 elaborates on SAND's capabilities for spatial data processing. Section 5 concludes the paper.

2. The SAND kernel

SAND consists of a kernel that implements basic objects and functions and a library that is responsible for assembling these into plans for evaluating higher level queries. Presently, the kernel implements atomic objects of common non-spatial types (strings and numbers) as well as a few choice two-dimensional geometric types (points, line segments, polygons, axes-aligned rectangles and regions). These objects are organized in tuples and tables using a relational-like data model. In order to access the functionality of the kernel in a flexible way, we opted to provide an interface to it by means of an interpreted language. We chose *Tcl* [17] for that role, mainly because it offers the benefits of an interpreted language but still allows code written in a high-level compiled language (in our case, C++) to be incorporated via a very simple interface mechanism.

A table, as implemented by SAND, is an abstraction which can better be defined by its functionality. In object-oriented programming, the concept known as *class* is used to refer to software components (*objects*) which share the same functionality. Objects of class *table* are repositories of data that can be handled one tuple at a time by means of a uniform set of functions, or, to use object-oriented terminology, *methods*. The SAND kernel implements a few table varieties (classes) and the SAND library implements a few more. In order to perform operations on any given table, that table must be "opened" -- a process not unlike opening a file. Table varieties implemented by the kernel are opened by using the `sand open` command, while those implemented by the library are opened via a call to a procedure. Either way, the result of such an operation is an open table "handle" which can be used to invoke table methods. Each opened instance of a table contains a memory buffer large enough to contain one tuple of that table, to which we refer here as a *tuple buffer*. A handle of an open table in SAND responds to the following set of methods:

Proceedings of the Seventh International Workshop on Spatial Data Handling, Delft, The Netherlands, August 1996

Spatial Database Programming Using SAND[1]

CLAUDIO ESPERANÇA
Universidade Federal do Rio de Janeiro
COPPE, Programa de Engenharia de Sistemas e Computação
Caixa Postal 68511, Rio de Janeiro, RJ 21945-970, Brazil
*e-mail:*esperanc@cos.ufrj.br

and

HANAN SAMET
Computer Science Department and Center for Automation Research and
Institute for Advanced Computer Studies, University of Maryland
College Park, Maryland 20742
*e-mail:*hjs@umiacs.umd.edu

Abstract

SAND (Spatial and Non-spatial Data) is an interactive environment that enables the development of spatial database applications. It was designed as a tool for rapid prototyping of algorithms and query evaluation plans dealing with spatial and non-spatial data. In this paper we give an overview of SAND's architecture and illustrate how typical spatial and non-spatial queries can be processed by means of short code fragments.

Keywords: Spatial databases, GIS, query optimization.

1. Introduction

The design of spatial database applications involves many stages. The first stage is choosing a proper development environment which entails the appraisal of the many existing software packages in which will supply the basic facilities needed for the task. The software components most commonly used in such applications are programs or libraries specialized in performing operations on spatial data, while non-spatial data is frequently handled by database management systems (DBMS). In fact, this combination has become so common that much effort has been put into integrating these components into a single framework. For the most part, these efforts have concentrated either in adding spatial capabilities to existing standard DBMS's [1, 2, 15, 21, 23] or in developing new database systems with the "spatial" aspect in mind [13, 16].

Our own experience with the problem [2, 3] has shown us that the so-called "extensible"

[1] The Support of the National Science Foundation under Grant IRI-92-16970 and of Conselho Nacional de Desenvolvimento Científico e Tecnológico is gratefully acknowledged.