To Appear in the Journal of Visual Languages and Computing

Experience with SAND-Tcl: A Scripting Tool for Spatial Databases [1]

CLAUDIO ESPERANÇA (Contact Author)
COPPE, Engenharia de Sistemas e Computação
Universidade Federal do Rio de Janeiro
Ilha do Fundão, CT, Bloco H
Rio de Janeiro, RJ 21949-900, Brazil
e-mail: esperanc@cos.ufrj.br Tel/Fax: (21) 590-2552

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 Tel: (301) 405-1755 fax: (301) 314-9115

Abstract

The use of scripting makes it possible to overcome many important difficulties in the development of database applications. By extending a general-purpose scripting language with constructs derived both from the database kernel and from the intended application domain, issues such as query processing and user interfacing can be approached in an economical and flexible way. This is illustrated by describing our experience with *SAND-Tcl*, a scripting tool developed by us for building spatial database applications. SAND-Tcl is an extension of the Tcl embedded scripting language with the constructs of the SAND environment for developing applications involving both spatial and non-spatial data. SAND-Tcl acts as a "glue" to hold together all the subsystems of SAND. In fact, query evaluation plans are SAND-Tcl programs (or *scripts*) which are written on-the-fly by SAND in response to a query defined by the user. This permits the rapid prototyping of algorithms and makes SAND a useful tool both for applications and research. The focus is on data storage, retrieval operations, and spatial indexing. Implementations of operations such as spatial selection, ranking, and spatial join are given. In addition, tools are described to make possible the construction of graphical user interfaces to a spatial database as well as providing users the ability to view and interact with spatial objects in a graphical manner. This is achieved through the use of SAND-Tcl scripts and the Tk graphical user interface toolkit which is tightly coupled to Tcl.

1 Introduction

A spatial database is essentially a software environment where large collections of spatial and non-spatial objects can be manipulated and queried. Of course, any concrete definition of what actually constitutes a spatial database is necessarily skewed by personal interpretations. Nevertheless, the following definition due to Güting [14] is fairly typical — that is, a spatial database system is a database system that:

1. offers spatial data types in its data model and query language, and
2. supports spatial data types in its implementation, providing at least spatial indexing and efficient algorithms for spatial join.

The advantage of using a conventional database management system (DBMS) is that the user has at his disposal a time-tested and presumably efficient way of retrieving and storing data. In addition, query languages often provide a rich set of query constructs that alleviate the necessity of further processing

[1]The support of the National Science Foundation under Grants IRI-9712715 and EIA-99-00268, the Department of Energy under Contract DEFG0295ER25237, and of Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) are gratefully acknowledged.

and thereby obviate the need for user-defined extensions which could be provided by a scripting facility. Moreover, query languages are designed with the goal of hiding implementation details from the user. Unfortunately, this may pose a problem if the database system itself has not been extended to support spatial data [34]. In this case, a spatial engine must be supplied in order to store and/or process spatial data. There are a number of levels at which the capabilities of the spatial engine and those of the DBMS may be integrated. These levels are characterized as *architectures* by Vijlbrief and van Oosterom [36] who divide them into three three major categories[2]:

1. *Dual architecture.* Two separate subsystems are used for storing spatial and non-spatial data. Non-spatial data is usually stored in a relational database management system (RDBMS), while spatial data is stored and processed by means of a proprietary spatial subsystem. The main problem with this approach is the lack of tight integration between the components responsible for processing spatial and non-spatial data. In particular, the construction of query evaluation plans (i.e., strategies for computing query answers) is hindered by the the fact that data which belongs logically to a single tuple or object is stored in two separate subsystems. For example, the ARC/INFO system [26] follows this approach.

2. *Layered architecture.* All spatial information is stored in a RDBMS by translating it into the available data types. Thus, for instance, a point in two dimensions would be represented as two floating point attributes, and a polygon as a relation of points. One problem inherent to this scheme is that it may be necessary to use costly database operations even to access a single object. A more serious problem, however, is that spatial access methods cannot be seamlessly integrated into the RDBMS. A typical example of this kind of architecture is SIRO-DBMS [1].

3. *Integrated architecture.* Systems where spatial and non-spatial data are handled at the same level by the database system. For example, points and polygons can be represented and operated upon in the same way as integers or strings. Similarly, spatial access methods are viewed as analogs to linear indices such as B-trees. Among others, the Gral [13], GEO++ [36], PARADISE [31], and SAND systems [10] all follow this approach.

From the above, it is fairly clear that the integrated approach is the most flexible. The SAND system, built by us, is a prototype environment for developing applications involving both spatial and non-spatial data based on the integrated approach. The SAND kernel implements a relational data model extended with several geometric functions and predicates as well as a number of access structures. In addition, SAND provides a library which permits a modular construction of query evaluation plans.

In contrast with other spatial database systems, however, SAND was not designed to be accessed from within a system programming language such as C or C++. The main interface to SAND is through SAND-Tcl, an extension to *Tcl* [30] which is an embedded interpreted scripting language. SAND-Tcl acts as a "glue" to hold together all the subsystems. In fact, query evaluation plans are SAND-Tcl programs (or *scripts*) which are composed on-the-fly by SAND in response to a query defined by the user. This is in contrast with the typical approach to query processing which is centered on a query optimizer buried fairly deep inside the DBMS engine. Even when such optimizers are built with extensibility in mind (e.g., Volcano [11] or OPT++ [21]), the task of adding a new query processing strategy or refining an existing one may entail considerable work. On the other hand, a SAND-Tcl application can easily combine both plans generated by the optimizer and custom-made plans. This is an important contribution of our work.

In this paper we describe SAND-Tcl and show how it is used to achieve an integrated architecture. In particular, we discuss our experience by presenting the solutions that we adopted by extending Tcl with the constructs of the SAND spatial database system. Our focus is on data storage, retrieval operations, spatial indexing and graphical user interfaces. Although a full-fledged database management system (DBMS) must

[2] Although this classification was originally intended for GISs, it is applicable to spatial databases as well.

also address other important issues such as concurrency control, security, and transaction management, we do not do this here as they are beyond the scope of this paper. Our aim is to show the utility of a scripting language by obtaining a suitable set of scripting language extensions to build applications for querying spatial databases and presenting the retrieved information in a graphical manner. A couple of examples, with different complexities, of such an application are presented in Section 7. These are scaled-down versions of the more generic SAND browser [10]. Note that our notion of browsing is one of spatial objects, rather than images (e.g., [32]). Our ultimate goal is to be able to build a Geographic Information System (GIS) using only scripts written in this extended language.

The rest of this paper is organized as follows. Section 2 reviews the scripting concept. Sections 3–5 describe how SAND-Tcl enables the realization of the definition of a spatial database [14] given in the opening of this section. In particular, Section 3 discusses the incorporation of an elementary database organization and its operations into SAND-Tcl. Section 4 deals with the treatment of spatial data types in SAND-Tcl. Section 5 shows how SAND-Tcl handles indexes including spatial indexes and describes some constructs that make use of them to facilitate the implementation of spatial queries. Section 6 discusses how scripting can be used to generate query evaluation plans on-the-fly. Section 7 overviews some of the tools provided by SAND-Tcl to enable the user to interact with the spatial objects in a graphical manner and presents a pair of examples. Section 8 briefly describes the *SAND Browser*, a full application built under the SAND-Tcl environment. Section 9 contains concluding remarks.

2 The Scripting Concept

Until recently, the most common way of interacting with operating systems and most computer applications was through text-based interfaces. The program which provides the most direct interface with the operating system, termed a *shell*, would then take user input (called *commands*), and arrange for the appropriate action to be executed. Those actions would either be performed by the shell program itself or by another application that would be invoked for that purpose.

Some shell programs became sophisticated to the point of allowing users to store command sequences (called *scripts*) that could be invoked on request, and even providing programming facilities such as variables and control structures to enable users to control automatically which actions were needed and in what order they should be performed. In fact, some computer applications were split into several separate programs and relied on shell scripting tools to combine these programs to perform a given task.

Today, textual interfaces are becoming less and less common, being gradually replaced by graphical user interfaces (GUI). Thus, modern shells and interactive computer applications are mostly controlled by means of graphical elements such as windows, icons, menus and pointers (called *WIMP*s [24]). However, such interfaces are not designed to allow users to specify complex sequences of operations in the same manner as shell scripts. The rationale is that users are not supposed to *program* — this activity is considered to be restricted to application developers. This has led to the creation of features such as keyboard macros that are intended to provide some means of customization of frequently repeated operations.

In some sense, scripting can be viewed as an activity that lies somewhere between application development and end-user utilization. Thus, a script can be produced by an application developer in order to provide some additional functionality, or it can be written by an end-user in order to automatically perform tasks that he or she has learned to do interactively.

Scripting does not conflict with the use of GUIs; instead, the two are complementary. Perhaps the best example of synergy between scripting and GUIs is the combination between the scripting language known as *Tcl* [30], and *Tk* [30], a graphical interface toolkit. Thus, one may write a *Tcl* script, say, to remove unwanted files in a given directory and, at the same time, use *Tk* to provide a graphical user interface that will allow the user to confirm the operation or to specify which files are not to be removed.

Some scripting languages (e.g., Python [23], Scheme [7], Tcl [30]) can be used to access code written in high-level compiled languages such as *C* or *C++*. In these cases, scripting languages are sometimes referred to as *embedded* languages. The idea is that while critical, computationally-intensive parts of an application need to be implemented as efficiently as possible (i.e., compiled), other parts can take advantage of the more flexible programming environment provided by an embedded interpreted language. The mechanism for this hybrid environment consists of extending the embedded language with commands (or other syntactic structures) which are bound to compiled code. Thus, an application developed under such an environment consists of a script that is executed by an extended interpreter.

Scripting plays a central role in SAND. Applications are written in SAND-Tcl by making use of:

1. Native Tcl/Tk commands.
2. The Tcl command interface to the SAND kernel.
3. Scripts stored in the SAND library.

This arrangement is shown in Figure 1. The SAND kernel implements the basic access methods needed to access the tables and indices (both linear and spatial) of the database. The SAND library implements higher-level functionality such as query evaluation plans and a (still incipient) query optimizer. An important aspect of scripting languages is that they enable the construction of code on-the-fly. In other words, an interpreted environment usually allows for scripts that have the ability to write other scripts. The SAND optimizer makes use of this facility in order to generate query evaluation plans in response to a query string expressed in a form that is similar to the well-known "select-from-where" construct of SQL. We return to this subject in Section 6.

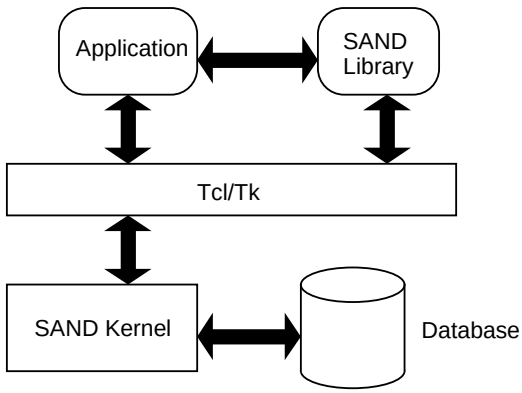


Figure 1: Components of the SAND system. Rounded boxes denote scripts and square boxes denote components written in a system programming language (C/C++).

It is important to emphasize at this point that the extended interpreter already provides most of the functionality needed to develop not one, but many applications, and hence it is already a useful product in its own right.

3 Elementary Database Organization and Operations

In SAND, storage access methods were designed according to a relational-like data model. Relations, linear indices, and spatial access methods are all regarded as specialized *tables* with different access structures. For example, let us define a relation called `city` for storing attributes of cities around the world. `city`

is composed of attributes `name` (a string of no more than 20 characters), `population` (an integer) and `location` (a point). This relation can be created from within a SAND-Tcl script by issuing the following command:

```
sand create city name char(20) population integer location point
```

Attribute `location` in the command above represents a spatial attribute (type `point`), where the *x* and *y* coordinate values represent, say, latitude and longitude. Notice that `location` is handled in much the same way as the other non-spatial attributes (`name` and `population`). A linear index in the form of a B-tree [8] and a spatial index implemented as either a PMR-quadtree [33] or an R-tree [16][3] can also be specified for relation `city` using a similar syntax. For example, the two commands below create linear index `cityname` for attribute `name` and spatial index `cityloc` for attribute `location`:

```
sand createindex cityname city name
sand createindex cityloc city location
```

In order to perform input and output on any table type, SAND provides a set of common commands which allow tuples to be read or written one at a time. Every time a table is opened for I/O, a buffer (called a *tuple buffer*[4]) for holding one tuple of that table is allocated. As a result of an *open* operation, a new operator (i.e., a command) called a *handle* is created on-the-fly in order to access the opened table or its associated tuple buffer. Notice that several opened instances of the same table may be active at a time, each referenced by a different *handle.* Figure 2 summarizes the command interface to general tables.

Common table access commands are illustrated in the following script which prints all tuples of relation `city` (i.e., the equivalent to SQL query *"select * from city"*):

```
1.  # Script to list all tuples of city
2.  set cityHandle [sand open city]
3.  set status [$cityHandle first]
4.  while {$status} {
5.      puts [$cityHandle get]
6.      set status [$cityHandle next]
7.  }
8.  $cityHandle close
```

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

- A pound sign (`#`) used at the beginning of a line denotes a comment.
- A dollar sign (`$`) is used for variable dereferencing. Thus, construct `$`*var* denotes the value of variable *var*.
- `set` *var value* assigns *value* to variable *var*. Variable *var* is created and storage is allocated for it if it did not exist before execution of the command.
- The construct `[` *string* `]` used as a value means that *string* is to be evaluated as a command and the result used instead.
- `puts` *string* is one of *Tcl*'s built-in commands that prints *string* on the standard output device.

A walkthrough of the script given above will help to make these concepts clear:

[3]The choice of which spatial index should be used is done by means of a command which is not explained here since it is not relevant to our discussion. It suffices to say that both structures provide exactly the same functionality.

[4]Memory is dynamically allocated for relations with variable-sized attributes such as polygons. Whereas this could lead to excessive use of main memory (e.g., for unusually large polygons), this simple policy is adequate for typical data sets.

Command	Semantics
`sand open` *name*	Open table *name* and return as its value a table handle (denoted by *tableHandle* in the commands below) which is a pointer to a structure that describes a new open instance of table *name*. In particular, *tableHandle* gives access to the tuple buffer for use in subsequent I/O operations.
tableHandle `close`	Close the open table with tuple buffer *tableHandle.*
tableHandle `first` *options*	Load the tuple buffer of *tableHandle* with the first tuple of the corresponding table. One or more *options* may be specified, which will alter the order or impose selection conditions for retrieving tuples. The available *options* depend on the type of the table. Returns true or false to indicate whether or not the operation was successful.
tableHandle `next`	Load the tuple buffer of *tableHandle* with the next tuple of the corresponding table. If *options* were specified in the corresponding `first` command, then they are respected implicitly in fetching the next tuple. Returns a true/false success code.
tableHandle `tid`	Return the tuple identifier, i.e., an integer number that uniquely identifies the tuple currently loaded in the tuple buffer of *tableHandle.*
tableHandle `get`	Return a list with the values of all attributes in the tuple buffer of *tableHandle.*
tableHandle.attr `get`	Return the value of attribute named *attr* in the tuple buffer of *tableHandle.*
tableHandle `assign` *value ... value*	Replace the values of the attributes in the tuple buffer of *tableHandle* with the given *value*'s in the same order specified in the table schema.
tableHandle.attr `set` *value*	Replace the contents of attribute named *attr* in the tuple buffer of *tableHandle* with the given *value.*
tableHandle `insert`	Add the contents of the tuple buffer of *tableHandle* as a new tuple in the table corresponding to *tableHandle.*
tableHandle `delete`	Delete the most recently accessed tuple from the table corresponding to *tableHandle.* The subsequent content of the tuple buffer is undefined.

Figure 2: Summary of common SAND-Tcl table commands.

1. The first line is a comment.

2. In line 2, variable `cityHandle` is created and initialized with the table handle obtained by executing `sand open city`. All subsequent commands that affect the just opened table will be invoked by commands starting with `$cityHandle`.

3. In line 3, the first tuple of `city` is fetched from disk by command `$cityHandle first`, which returns a Boolean code denoting the success of the operation. This code is stored in variable `status`.

4. Lines 4 through 7 implement a loop to fetch and print consecutive tuples of `city`. The loop is controlled by the value of variable `status`. Hence, when the fetch operation in line 6 fails, the loop is terminated.

5. Line 8 closes table `city` as it is the one whose tuple buffer is bound to `$cityHandle`.

In addition to the common table commands, each table type supports additional table commands that reflect their particular access capabilities. Thus, relations can be referenced by tuple identifier (i.e., *tid*) using the `fetch` command, a linear index can be accessed by its contents using the `find` command, and spatial indices can be accessed according to several geometric constraints.

As an example of the usage of table access commands, consider procedure `city-location` which, given the name of a city as an argument, returns the city's location. The strategy for computing the answer to this query involves the use of linear index `cityname` defined on attribute `name` of `city`. In summary, procedure `city-location` implements an evaluation plan for a query that would be expressed in SQL as *"select location from city where name=argName"*):

```
1.  # Returns the location of city  $argName
2.  proc city-location { argName } {
3.      set cityHandle [sand open city]
4.      set nameHandle [sand open cityname]
5.      $nameHandle find $argName
6.      $cityHandle fetch [$nameHandle tid]
7.      set result [$cityHandle.location get]
8.      $cityHandle close
9.      $nameHandle close
10.     return $result
11. }
```

Notice that in line 5 the linear index is used to find a city in index `cityname`, whereas the corresponding tuple of `city` must be fetched with a `fetch` command (line 6). This is the mechanism used in SAND-Tcl to establish the connection between indices and relations, i.e., index records store the tuple identifiers of the corresponding relation records.

4 Spatial Data Types

The term "spatial data type" can be used to refer to several different data abstractions. It is reasonable to assume that a spatial datum is a value that denotes at least a location in space, but frequently is tied to other spatial notions such as proximity, extent and shape. Thus, if we restrict this discussion to two-dimensional space, we may take for granted that simple geometric objects such as points, lines and polygons are to be considered "spatial data types". However, it is not clear what level of aggregation between simple data types should be supported by a spatial database. For example, a polygonal map, i.e., a partition of space into non-overlapping adjacent polygons, may be considered a "complex" spatial data type. Of course, if the database supports polygons as a simple type, then a polygonal map may be represented by a set of polygons, but this representation scheme must be enforced at all times. In other words, not all sets of polygons represent valid polygonal maps and care must be taken to avoid database operations that might produce an invalid configuration of polygons.

Overall, when defining the data model for spatial information in the context of a DBMS, we must consider what types of operations are to be performed on such data and how these operations affect the retrieval of data from disk. For instance, if polygons are to be supported as a spatial attribute type in the context of a relational DBMS, then they may be stored in variable-length records along with other fixed-size

attributes. Alternatively, we might choose to keep polygons in a secondary disk-based data structure and store only pointers to them in the relation.

In the SAND system, primitive spatial data types are supported in the form of spatial attributes. Thus, for instance, a polygonal map can be represented as a table where one of the attributes is of type `polygon`. New spatial types can be incorporated into the system by means of an extension mechanism. SAND already supports some of the most common primitive geometric objects in two dimensions: `point`, `line`, `rectangle`, `polygon`, `polyregion` and `region` (see Figure 3). For example, relation `city` defined earlier contains spatial attribute `location` which is of type `point`.

Type Name	Semantics
`point`	A point.
`line`	A line segment.
`rectangle`	An axes-aligned rectangle (i.e., its edges are parallel to the coordinate axes).
`polygon`	A simple polygon, i.e., consisting of a single non-intersecting contour.
`polyregion`	An arbitrary polygon containing any number of contours or holes.
`region`	An arbitrary orthogonal polygon, i.e., a polygon containing any number of contours or holes, but whose edges are parallel to the coordinate axes.

Figure 3: Primitive spatial data types supported in SAND.

The SAND storage model employs one of the following two strategies when handling attribute types which are variable-sized, namely the `polygon` and `region` types. They may be stored in tables whose records may have variable size, or they may be stored in separate disk files (termed *containers*). If a table includes an attribute which is stored in a container file, then each tuple contains only a bounding structure (e.g., a minimum bounding rectangle) and a pointer for that attribute's value in the container file. The advantage of this approach is that the full value of the spatial attribute is only accessed if there is need for it (this is also known as the *filter-and-refine* strategy [29]). For example, if a query consists of selecting tuples based on the value of a non-spatial attribute, then only the tuples that satisfy the condition will have the attribute values stored in containers retrieved. On the other hand, accessing each tuple that satisfies the query involves two disk accesses: one access to retrieve the non-spatial attributes and the pointer to the spatial attribute, and another access to retrieve the spatial attribute proper.

Another aspect that must be considered is the set of operations involving spatial data that are to be supported. Some common operations are:

- Operations that compute integral properties such as *area*, *perimeter*, or *centroid*.
- Geometric operations which result in another spatial object. These include, among others:
  - Computing the convex hull of a polygon or its minimum bounding rectangle.
  - Set operations such as computing the intersection of two line segments or the union of two polygons.
- Operations that determine relationships between two objects. These can be divided into the following categories [9]:

– *Topological relationships* such as determining whether two objects touch, intersect, contain or are contained by each other.
– *Directional relationships* which try to ascertain the relative location of two objects such as whether one is above or to the left of the other.
– *Metric relationships* such as the distance between two objects.

SAND-Tcl provides a command interface for the three spatial operations which may be used on any spatial data type in SAND: `bbox`, `intersects`, and `distance` (see Figure 4). Additional operations are supported for some spatial data types. For example, the `rectangle`, `polygon`, `polyregion` and `region` types support the `perimeter` operator which returns the length of their boundaries. The SAND kernel currently implements only a small set of spatial data types and operations which are sufficient for most GIS-related applications. The implementation of a more comprehensive set of spatial data types such as that defined by the ROSE Algebra [15] is under consideration.

Command	Semantics
`bbox` s_1 ... s_n	Return the minimum bounding rectangle that encloses all given spatial objects.
`intersects` s_1 s_2	Return 1 (*true*) if s_1 and s_2 intersect, i.e., have at least one point in common.
`distance` s_1 s_2	Return the Euclidean distance between s_1 and s_2.

Figure 4: SAND-Tcl commands for spatial operations supported for all spatial data types in SAND.

The scripting environment of SAND-Tcl allows the relatively small set of built-in spatial operations in SAND to be combined so as to express many common predicates. For example, consider the predicate `south-of` *point feature*. it evaluates to *true* if *feature* (a spatial attribute of any type) consists only of points whose *y* coordinate values are less than the *y* coordinate value of *point*. Another way of expressing this predicate is that it evaluates to *false* if there is any point of *feature* with *y* coordinate values which are greater than or equal to *point.y* (i.e, the *y* coordinate value of *point*). A procedure `south-of` that implements this predicate in SAND-Tcl can be easily constructed:

```
1.  proc south-of { point feature } {
2.      set northRect [bbox $point "line -1e9 1e9 1e9 1e9"]
3.      if {[intersects $feature $northRect]} {
4.          return 0
5.      }
6.      return 1
7.  }
```

In line 2 of this script, variable `northRect` is assigned a rectangle with its bottom side touching *point* which is wide and tall enough to approximate the semi-space given by equation $y \geq point.y$. This is ensured by computing the bounding box of *point* and a line segment defined between points $(-10^9, 10^9)$ and $(10^9, 10^9)$[5] (see Figure 5). A feature is not `south-of` *point* if it intersects `$northRect` (lines 3 and 4); otherwise, procedure `south-of` returns 1 (i.e, *true*).

[5]It is clear that values -10^9 and 10^9 are used here to represent approximations of minus and plus infinity, respectively. In practice, these values would be computed by taking into account the nature of the application. If spatial values represent geographical entities expressed in latitude-longitude format, then -180 and $+180$ (degrees) would suffice.

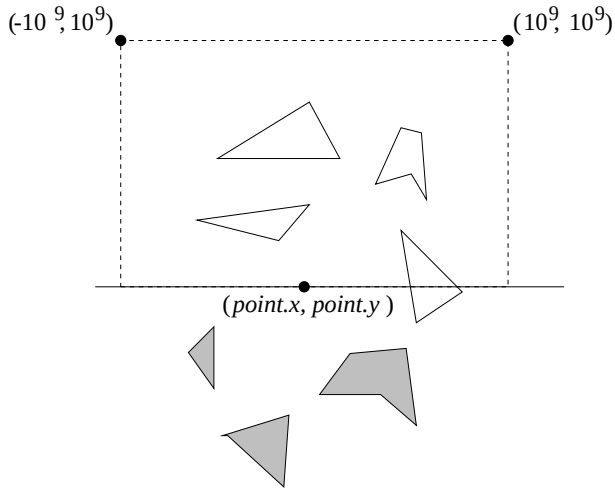


Figure 5: Computing predicate `south-of` *point*. Only features that do not intersect the rectangle drawn with broken lines satisfy the predicate (gray objects).

5 Indexing

The term "index" when used in the context of a database is a structure designed to facilitate accessing the data based on its contents. When the datum being searched is non-spatial, the constraints for the search usually consist of requiring the data to fall within a user-specified range. If the operation is restricted to one-dimensional domains, then the index is said to be *linear*. Thus, if we consider our example relation `city`, then linear index `cityname` can be used to quickly find "Berlin" or even all cities whose names start with "B".

A linear index is frequently implemented as a tree structure which maintains the indexed data in sorted order. B-trees and its variations [8] are the most widely used forms of linear indices. Thus, in order to locate a value, the tree is traversed from the root until encountering one of its leaves which holds the sought value. Ranges of values can be similarly retrieved by using the tree structure to find neighboring leaves. The ordering induced by such structures becomes apparent in the behavior of commands `first` and `next` when applied to open linear index tables. For example, if *nameHandle* is an open instance of table `cityname`, then *nameHandle* `first` and *nameHandle* `next` can be used to retrieve all cities in alphabetical order.

When dealing with spatial data, indexing assumes a somewhat different perspective since it is not possible to impose a total ordering on the values. Queries on spatial data are based on *proximity*, i.e., they consist of criteria to select objects which are close to locations designated, say, by a point or a region. Data structures designed to cope with this problem are known as *spatial indices* or *spatial access methods.* The overall approach employed in the design of such structures is to strive to satisfy the property that objects which are close in space are also "close" within the scope of the data structure. Some data structures, such as the grid file [27] and the k-d-tree [5] are especially useful when handling points. Other data structures are suited to handle not only points, but also spatial objects with extent such as line segments or polygons. For example, the R-tree [16] and some of its variants [4, 35] as well as quadtree-based data structures [33] have been found useful in spatial database environments.

Some of the most common types of spatial selection queries are:

- *Intersection query:* Find all objects that intersect (i.e., have at least one point in common with) a spatial object given as an argument (termed a *query feature*). If the query feature is an axes-aligned rectangle, then this operation is also known as a *window query* or a *range query*.

- *Containment query:* Find all objects that are contained in a query feature. This is similar to the intersection query except that the entire extent of the sought objects have to be enclosed within the query feature. Clearly, when handling point data objects, containment and intersection queries are equivalent. Also, since intersection is a requirement for containment, algorithms and spatial indices that support intersection queries can be easily adapted for to handle containment queries by performing additional containment tests.

- *Distance query:* Find all objects that are within a given distance of a query feature. In such queries, "distance" usually means Euclidean distance, but other metrics can also be used, e.g., the city-block (or Manhattan) metric. Notice that distance queries may be viewed as special cases of intersection queries where the distance is required to be zero (regardless of what metric is used).

- *Ranking query:* Sort all the objects in order of their distance from a query feature. This is a generalization of the *nearest neighbor query* which is only interested in the first object of the sorted list. When ranking is performed on all existing objects, spatial indices are of limited value since this operation can be performed by computing the distances of all objects to the query feature and then sorting the results using conventional methods. However, if the search is to be done incrementally, spatial indexing becomes indispensable. An incremental ranking operation should determine the $(k+1)^{st}$ closest object by gathering a minimum amount of additional information after having computed the k^{th} closest object. In particular, a good implementation of incremental ranking should determine the closest object in the same amount of time as a well-designed algorithm for computing nearest neighbor queries. Thus, to compute the $(k+1)^{st}$ nearest neighbor, it should not have to compute all $k+1$ neighbors. In a sense, incremental ranking can be regarded as the most general spatial selection query since both intersection and distance queries can be obtained trivially by examining the objects reported by the ranking operation.

In SAND, spatial indices are tables which can be accessed by means of a PMR-quadtree [33] or an R-tree [16]. All spatial selection queries listed above are supported by augmented forms of the `first` command. Incremental ranking operations are supported by using the algorithm described in [17, 19]. The behavior of the `first` command can be modified to accommodate spatial selection criteria by means of *options* (refer to Figure 6). Issuing a `first` command, possibly accompanied by one or more spatial selection *options*, initiates a new scanning process of the corresponding table. Thus, subsequent `next` commands may be used to retrieve all tuples which satisfy the conjunction of all conditions expressed in the *options*. For example, the following script can be used to retrieve all tuples of `city` in the northern hemisphere that are not farther away from "Paris" than "New York"[6] in the order of their distance from "Paris".

```
1.  set parisLoc [city-location "Paris"]
2.  set cityHandle [sand open city]
3.  set locHandle [sand open cityloc]
4.  set northRect {rectangle -1e9 0 1e9 1e9}
5.  set status [$locHandle first -ranking $parisLoc -intersects $northRect]
6.  while {$status} {
7.      $cityHandle fetch [$locHandle tid]
8.      if {[$cityHandle.name get]=="New York"]} break
9.      puts [$cityHandle get]
10.     set status[$locHandle next]
11. }
12. $cityHandle close
```

[6] We assume that both "Paris" and "New York" are in the database.

Option	Semantics
`-ranking` *feature*	Denotes an incremental ranking operation. Tuples will be retrieved in increasing order of distance from *feature*, unless option `-farthest` is also specified in which case tuples are returned in decreasing order of distance from *feature*.
`-min` *distance*	Restricts the ranking operation so that tuples which are closer than *distance* units from the ranking feature are not considered. Must be specified together with a `-ranking` option.
`-max` *distance*	Restricts the ranking operation so that tuples which are farther than *distance* units from the ranking feature are not considered. Must be specified together with a `-ranking` option.
`-farthest`	Inverts the scanning order so that tuples are returned in decreasing order of distance from the ranking feature. Must be specified together with a `-ranking` option.
`-within` *distance feature*	Restricts the scanning to tuples whose distance to *feature* is no greater than *distance*. *feature* is usually different from the ranking feature.
`-intersects` *feature*	Restricts the scanning to tuples which intersect *feature*. *feature* is usually different from the ranking feature.
`-contains` *feature*	Restricts the scanning to tuples which contain *feature*.

Figure 6: Options for command `first` when applied on spatial indices.

```
13. $locHandle close
```

A few observations about the script given above:

- In line 1, the location of Paris is obtained using the procedure `city-location` presented in Section 3.
- In line 4, variable `northRect` is set to a rectangle that represents the northern hemisphere. This is similar to the computation of variable `northRect` in procedure `north-of` (see Section 4).
- In line 7, the tuple of `city` that corresponds to the current tuple of *cityloc* is retrieved.
- It is not necessary to know the location of "New York" before the scan of spatial index `cityloc` is initiated in line 5 since the tuples will be retrieved in order of distance from "Paris". Thus, when the tuple that corresponds to "New York" is reached, all cities that are closer to "Paris" will have been reported already, and the search can be terminated. This test is made in line 8.

Spatial indices may also be used in the implementation of queries known as *spatial joins*. A spatial join may be regarded as a generalization of a spatial selection where the query feature is an attribute of a second relation. For example, if we assume the existence of a relation called `country` with attributes `name` (the name of the country) and `area` (the region occupied by the country), then an example of a spatial join is an operation that seeks to find all cities lying within each country. Notice that in order to ascertain that a given

city lies within a country, it should be the case that the corresponding value of attribute `location` of `city` intersects the region given by attribute `area` of `country`.

Spatial joins are usually very costly to compute, especially in the absence of spatial access methods. For example, consider a spatial join query between relations `city` and `country` where we wish to compute all pairs of tuples $city_i/country_j$ where $city_i$ lies within $country_j$. This query may be decomposed into several spatial selection queries of the sort "*Retrieve all tuples from city where location intersects* $area_j$", in which $area_j$ corresponds to each value of attribute `area` of `country`. Clearly, if no spatial indices are defined, then the computation of each spatial selection query requires that each value of attribute `area` of `country` be tested against each value of attribute `location` of `city`. However, this situation may be improved with the use of spatial access methods. In particular, spatial index `cityloc` may be used in the evaluation of each spatial selection query in order to speed up the access to tuples of `city` whose `location` attribute intersect a given `area` value. This approach is merely an extension of the technique known as *index join* [22] in the context of conventional databases. The SAND-Tcl script below illustrates how the *index join* technique can be used in answering the spatial join query just described:

```
1.  set cityHandle [sand open city]
2.  set locHandle [sand open cityloc]
3.  set countryHandle [sand open country]
4.  set status1 [$countryHandle first]
5.  while {$status1} {
6.      set status2 [$locHandle first -intersects [$countryHandle.area get]]
7.      while {$status2} {
8.          $cityHandle fetch [$locHandle tid]
9.          puts "[$cityHandle.name get] is in [$countryHandle.name get]"
10.         set status2 [$locHandle next]
11.     }
12.     set status1 [$countryHandle next]
13. }
14. $cityHandle close
15. $locHandle close
16. $countryHandle close
```

If both data sets involved in the join operation are supported by spatial indices, then the query can be computed by traversing the indices in tandem. This strategy bears some resemblance to the technique known as *merge join* [22] employed in most DBMS's. Spatial join algorithms following this approach have been reported for R-trees in [6] and for PMR-quadtrees in [20]. Günther also describes an algorithm for in tandem traversal of data structures termed *generalization trees* in [12].

SAND also supports the *merge join* approach where two spatial indices are traversed in tandem [18]. Command `sand join` in SAND-Tcl initiates a join operation using a merge-join strategy. It has the following syntax:

`sand join` *tableHandle1 tableHandle2 option ... option*

where *tableHandle1* and *tableHandle2* correspond to open spatial index tables and *option* is zero, one or more options. The options are explained below.

The value returned by the `sand join` command is a *tableHandle* similar to that returned by the `sand open` command which is then assigned to be the value of a variable such as *joinHandle* , i.e., once the join operation is initiated, commands "*joinHandle* `first`" and "*joinHandle* `next`" can be used to access one pair of tuples at a time. In fact, the `sand join` command implements a variation of the incremental ranking operation, where pairs of tuples lying at increasing distances are visited at each step. In particular, pairs of

Option	Semantics
`-min` *distance*	Restricts the ranking operation so that tuples which are closer than *distance* units from each other are not considered.
`-max` *distance*	Restricts the ranking operation so that tuples which are farther than *distance* units from each other are not considered.
`-farthest`	Inverts the scanning order so that pairs of tuples are returned in decreasing order of distance.

Figure 7: Options for command `sand join`.

intersecting tuples are reported first. By using the `-min` and `-max` options, the process can be restricted to report pairs of tuples lying at any given range of distances. The options are summarized in Figure 7.

A sample usage of the `sand join` command is illustrated in the script below, which implements a spatial join between relations `city` and `country` by traversing spatial indices `cityloc` (a spatial index defined on attribute `location` of `city`) and `countryarea` (a spatial index defined on attribute `area` of `country`). The incremental ranking operation is restricted to the retrieval of pairs of values lying at distance zero from each other, so that the resulting effect is to retrieve pairs $city_i/country_j$ where $city_i$ is inside $country_j$.

```
1.  set cityHandle [sand open city]
2.  set countryHandle [sand open country]
3.  set locHandle [sand open cityloc]
4.  set areaHandle [sand open countryarea]
5.  set joinHandle [sand join $locHandle $areaHandle -max 0]
6.  set status [$joinHandle first]
7.  while {$status} {
8.      $cityHandle fetch [$locHandle tid]
9.      $countryHandle fetch [$areaHandle tid]
10.     puts "[$cityHandle.name get] is in [$countryHandle.name get]"
11.     set status [$joinHandle next]
12. }
13. $joinHandle close
14. $areaHandle close
15. $locHandle close
16. $countryHandle close
17. $cityHandle close
```

6 On-the-fly Plan Generation

Query optimizers (e.g., [3, 11, 21]) are prime examples of programs which generate other programs. The ultimate goal of a query optimizer is to produce a program that evaluates a given query in the most efficient manner. This program is called a *query evaluation plan*. A plan is usually constructed by combining several access strategies such as sequential scans, indexed sequential scans, nested loops, etc. Once a given combination of access strategies is chosen by the optimizer, a data structure must be created to represent the plan. Moreover, there must be some mechanism by which the actions represented by this data structure can be performed. If the programming environment does not allow for code to be handled in much the same

way as data, implementing this mechanism can be fairly complicated and awkward. On the other hand, the generation of plans in SAND is considerably simpler since a plan is simply a string which can be compiled later on-the-fly by the Tcl engine and executed.

For example, consider the *index join* script presented in Section 5 which computes all pairs of tuples $city_i/country_j$ where $city_i$ lies within $country_j$. Any index join plan generated by a query optimizer would probably follow the same overall structure used in that script, except that other tables, attributes or predicates might be involved. In short, a generalized index join plan might be expressed in a string containing the following text:

```
1.  set table2Handle [sand open TABLE2]
2.  set index2Handle [sand open INDEX2]
3.  set table1Handle [sand open TABLE1]
4.  set status1 [$table1Handle first]
5.  while {$status1} {
6.      set status2 [$index2Handle first SELECTOPTIONS]
7.      while {$status2} {
8.          $table2Handle fetch [$index2Handle tid]
9.          ACTION
10.         set status2 [$index2Handle next]
11.     }
12.     set status1 [$table1Handle next]
13. }
14. $table2Handle close
15. $index2Handle close
16. $table1Handle close
```

The above "template" script contains the essential control structures but several key elements (denoted by the all-uppercase symbols) still need to be replaced. For instance, in order to generate the index join script presented in Section 5, `TABLE1` must be replaced by `country`, `TABLE2` by `city`, and so forth. It is important to notice, however, that the technique of replacing key elements in a template script is substantially more powerful than, say, passing parameters to a procedure that implements the overall plan. For example, suppose that our example query is redefined to be: "Print all pairs all pairs of tuples $city_i/country_j$ where $city_i$ lies within $country_j$ AND the population of $city_i$ is 100,000 or more." This can be accomplished by replacing the `ACTION` key element with the following:

```
if {[$table2Handle.population]>100000]} {
    puts "[$table2Handle.name get] is in [$table1Handle.name get]"
}
```

In other words, key elements in a template script can be replaced not only by ordinary data but also by code fragments. SAND's query optimizer uses this same approach in order to build complex plans. In some sense, we may consider this process to be a form of rule-based optimization (see, for instance, [3]).

Once all the key elements of a template script are substituted, the resulting string comprises a fully functional plan. This string can then be evaluated, causing the plan to be executed, or stored for later use.

7 Scripts and Graphical User Interfaces

Scripting, as we have seen so far, provides an adequate means for expressing and implementing the crucial aspects of a spatial database application. Nevertheless, the text-based interface provided by the language

interpreter is too intimidating for the average user. Moreover, a proper visualization of spatial relationships requires that spatial objects be displayed graphically. Hence, our scripting environment must provide tools that not only make possible the construction of graphical user interfaces but also enable users to view and interact with spatial objects in a graphical manner. In accordance to the overall approach used in the design of SAND, the goal is not to provide a complete model for graphical user interaction (as described, say, in [37]), but rather to provide loosely-coupled tools that can be used within a script.

The construction of a graphical user interface usually presupposes a target platform environment such as Microsoft Windows [25], Apple Macintosh [2], or OSF/Motif [28]. Although all of these environments have many elements in common, each GUI offers a different set of capabilities, visual styles and interaction guidelines which constitute what is termed the GUI *look-and-feel.* More importantly, each GUI vendor provides a different programming interface, which would make the development of portable multi-platform applications a rather difficult task. Fortunately, however, *Tcl* already comes equipped with a scripting interface to *Tk* [30], a rather powerful GUI toolkit. In fact, both *Tcl* and *Tk* have been ported to many platforms providing native look-and-feel in most of them, including Microsoft Windows and Apple Macintosh.

In *Tk*, GUI components are called *widgets*. A widget of a given type is created by a corresponding *Tcl* command. For example, the `button` command creates a labeled button which, when pressed, performs a given task expressed by a script. Other component types include radio buttons, check buttons, menus, messages, textual entry widgets, graphical drawing widgets, etc. Each component creation command supports a variety of options that control several aspects of the component. For example, the `button` command supports, among others, option `-text` which is used to specify the text label to be displayed inside the button and option `-command` which specifies the script to be executed when the button is pressed.

Once the GUI components are created, they must be mapped to the screen by using what is called a *geometry manager* which controls their overall size and position relative to each other or to the window that contains them. *Tk* provides a number of geometry managers that differ in their policies for arranging components. One of the most commonly used geometry managers is known as the *packer* which is controlled by command `pack`.

The widget types provided by *Tk* are powerful enough for many applications. However, for our spatial database application we found a need for a widget for drawing spatial objects. This prompted us to create in SAND-Tcl a custom widget called `SandMapDisplay`, which can be used to graphically input and display all spatial types provided by SAND and to support other features such as panning and zooming. The command `SandMapDisplay` creates a new widget of that type (i.e., an instance of it). Its syntax is similar to that of most *Tk* widgets:

`SandMapDisplay` *mapName tableHandle option ... option*

where *mapName* is a symbolic name for the new instance of widget `SandMapDisplay`, and *tableHandle* designates an open table previously returned by `sand open` command. Zero, one or more *options* can also be specified to change display attributes such as colors, line width, etc.

Once a *SandMapDisplay* widget is created, it can be used to display the spatial attribute[7] of the open table given by the *tableHandle* argument. *mapName* identifies the newly created widget so that all operations on it are performed by commands prefixed by *mapName*. Figure 8 summarizes the most important commands on a `SandMapDisplay` widget named *mapName*.

As an example, consider the script given below, which implements a simple GUI consisting of three widgets for displaying tuples of relation `country` one at a time. The appearance of the GUI is illustrated in Figure 9.

[7] The `SandMapDisplay` widget assumes that each table contains only one spatial attribute in its schema. This is not a restriction of SAND, but we found that such an assumption simplified the design of the widget while not imposing too severe a limitation for most applications.

Command	Semantics
mapName **drawfeature** *feature*	Displays the spatial feature given by *feature*, e.g., the value of a spatial attribute or a constant such as **point 10 10**.
mapName **update**	Shorthand notation for the **drawfeature** command. The *feature* to be displayed in this case is the value of the spatial attribute currently loaded in the tuple buffer associated with widget *mapName* (i.e., the one bound to the *tableHandle* argument when widget *mapName* was created).
mapName **input***Type*	Enables the user to enter a spatial feature of the given *Type* (e.g., **inputpoint**, **inputline**, etc.) by drawing it with the mouse. The entered feature is returned as a result of the command.

Figure 8: Commands associated with widget `mapName` of type `SandMapDisplay`.

```
1.  set countryHandle [sand open country]
2.  $countryHandle first
3.  SandMapDisplay .display $countryHandle -bg white
4.  button .next -text "Next" -command {
5.      $countryHandle next
6.      puts [$countryHandle.name get]
7.      .display update
8.  }
9.  button .quit -text "Quit" -command exit
10. pack .display .next .quit
```

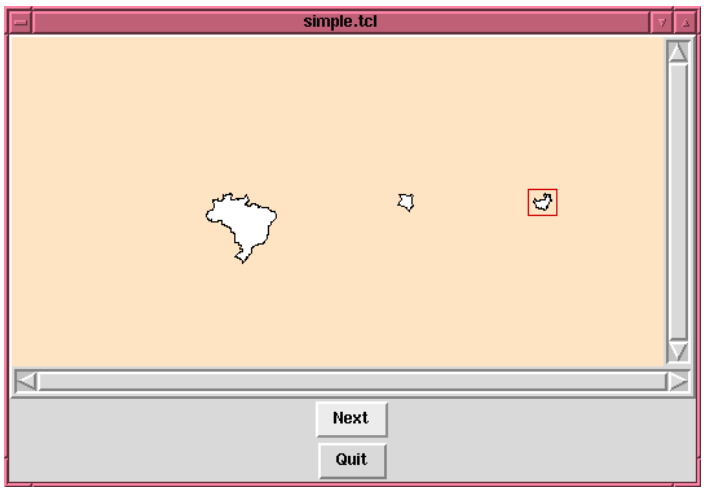


Figure 9: GUI of an application which displays countries of the world one at a time.

few observations about the script given above:

- Three widgets comprise the application GUI: A **SandMapDisplay** widget called **.display** (line 3), and two **button** widgets called **.next** and **.quit** (lines 4 and 9, respectively).

- The `-bg` option in line 3 specifies the color (`white`) used to fill polygons. In particular, countries are polygons which will be painted (filled) with that color.
- When widget `.display` is created (line 3), it automatically displays the tuple currently loaded in the tuple buffer of `$countryHandle`,that is, the first tuple of table `country` retrieved in line 2. In this particular data set, this corresponds to Brazil (the leftmost country in Figure 9).
- In line 4, option `-command` specifies the action that is to be performed each time the user presses the button labeled "Next". This is a script that extends to line 7 and consists of obtaining the next tuple of `country` (line 5), printing the country name (line 6) and displaying its area (line 7). Figure 9 shows the appearance of GUI after the button "Next" was pressed twice, returning Kenya and Indonesia. The `SandMapDisplay` widget is designed so as to automatically highlight the current value of the tuple buffer by means of an enclosing rectangle. In Figure 9, this corresponds to Indonesia.
- The `pack` geometry manager is used in this application. Since no additional information other than the widget names are specified in line 10, the default behavior of `pack` is to display widgets one below the other.

The example given above provides a GUI for a very simple query. A typical application would include a more natural graphical interaction and queries which are more complex. We conclude this section with a complete GUI application that implements spatial queries on the example relations `city` and `country`. The purpose of this application is to enable the user to interact with the digital chart of the world (DCW) and obtain the cities of each country he or she selects with the mouse. The script of this application is moderately long and is given in Figure 10. The interested reader will recognize the similarity existing between this code and the scripts presented earlier. Figure 11 shows the graphical user interface of the application where some of the cities of Brazil are displayed.

The window is divided into three panes described below, from top to bottom:

1. The main graphical display area implemented by a `SandMapDisplay` widget.
2. A `text` widget where the names of countries and cities are printed.
3. A `frame` widget containing three command buttons:
   (a) *First:* Pressing this button initiates a new query. Once this button has been pressed, the user should move the mouse over one of the countries displayed in the main area and then click the mouse button. This results in printing the name of the corresponding country, as well as that of the name of the closest city in the corresponding country to the location where the mouse was clicked. From now on, the cities (i.e., points) are ranked by pressing the "Next" button. Notice that each time the user presses the "First" button, a new query is started.
   (b) *Next:* Pressing this button retrieves the next closest city in the selected country to the location where the mouse was clicked, displays it on the main graphical area, and prints its name.
   (c) *Quit:* Terminates the application.

The script shown in Figure 10 is mostly built with constructs and commands described earlier in this paper. Nevertheless, it uses the following features of *Tk* which have not been discussed so far:

- The `text` widget named `.history` defined in line 10 is used to exhibit textual information in a GUI environment. As new countries and cities are retrieved, their names are appended to the display using the construct "`.history insert end` *string*" (lines 17, 21 and 27). Notice that "`\n`" (line 17) denotes a newline character.

```
 1.  # Open all tables
 2.  set cityHandle [sand open city]
 3.  set countryHandle [sand open country]
 4.  set locHandle [sand open cityloc]
 5.  set areaHandle [sand open countryarea]
 6.  # Generate all GUI components
 7.  # Main display pane:
 8.  SandMapDisplay .display $cityHandle -bg white -size 300
 9.  # Pane for printing country and city names:
10.  text .history -height 4
11.  # Command pane:
12.  frame .cmd
13.  button .cmd.first -text "First" -command {
14.     set pt [.display inputpoint]
15.     $areaHandle first -ranking $pt
16.     $countryHandle fetch [$areaHandle tid]
17.     .history insert end "Country:  [$countryHandle.name get]\n"
18.     $locHandle first -intersects [$areaHandle.area get]
19.     $cityHandle fetch [$locHandle tid]
20.     .display update
21.     .history insert end "Cities:  [$cityHandle.name get]"
22.  }
23.  button .cmd.next -text "Next" -command {
24.     $locHandle next
25.     $cityHandle fetch [$locHandle tid]
26.     .display update
27.     .history insert end ",[$cityHandle.name get]"
28.  }
29.  button .cmd.quit -text "Quit" -command exit
30.  # Map GUI components
31.  pack .display .history .cmd
32.  pack .cmd.first .cmd.next .cmd.quit -side left
33.  # Initialize the main display pane by drawing all countries
34.  set status [$countryHandle.area first]
35.  while {$status} {
36.     .display drawfeature [$countryHandle.area get]
37.     set status [$countryHandle.area next]
38.  }
```

Figure 10: SAND-Tcl GUI application to list and display cities of countries selected by the user.

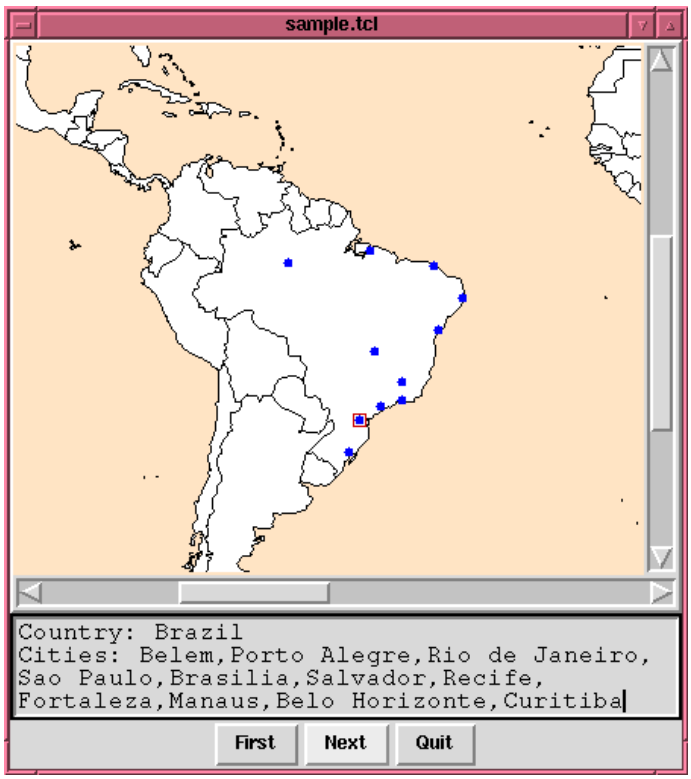


Figure 11: GUI of a sample application for querying cities and countries of the world given in Figure 10.

- A `frame` widget named `.cmd` is created in line 12 to enclose the command buttons `.cmd.first`, `.cmd.next` and `.cmd.quit`. The `pack` geometry manager is informed that these buttons are to be displayed side by side by means of option `-side left` (line 32).

8 The SAND Browser

The SAND-Tcl environment should not be regarded as merely a tool for writing short, disposable applications. Such a view is shared by many people who argue that this is the intended use of scripting environments, whereas "solid" or production-strength software should be developed entirely in standard compiled languages such as *C* or *C++*. Our claim, and experience, is that scripting environments can be used for "solid" production-strength software. To support our claim, we briefly describe the *SAND Browser*, the most complex application written under the SAND-Tcl environment. It presently consists of 3000+ lines of Tcl code and its purpose is to experiment with interaction paradigms for expressing and presenting spatial and non-spatial queries. Figure 12 presents a typical SAND Browser window displaying the (partial) result of a distance semi-join query [18].

3.2in

Distance join and distance semi-join queries cannot be directly expressed in SQL, but can be easily expressed by the user through the SAND Browser interface. The particular query result being displayed in Figure 12 involves two relations:

1. `city` which is the data set consisting of cities used in previous examples.

2. `facility` which contains the name and position (among other attributes) of all known nuclear facilities.

The query results in browsing the `city` relation by finding the closest element of the `facility` relation to each city. Successive city/nuclear facility pairs are obtained in the order of increasing distance between the elements of the pairs. Once the query has been run to completion, we have effectively found for each facility f, its closest cities so that these cities are closer to f than to any other facility $f\prime$ such that $f\prime \neq f$. This is a distance semi-join query where the semi-join is taken between the `city` relation and the `facility` relation. It should be noted that this query is closely related to the Voronoi diagram of the set of points defined by the `facility` relation — that is, the cities associated with a given facility f are those that fall within the Voronoi cell associated with f.

It is important to point out that the code of the SAND Browser script deals mostly with the aspects of graphical presentation and interaction. Critical and time-consuming code such as that required by the algorithm for computing distance joins or the algoritm for computing Voronoi diagrams was programmed in C++ and encapsulated into the SAND-Tcl interpreter. This code can be accessed by means of a very small set of Tcl commands. In fact, the analogous C++ interface, i.e. the set of C++ methods for accessing the same functionality is at least one order of magnitude more complex than the Tcl bindings. This arrangement proved to be beneficial, since the SAND Browser interface has been overhauled several times in order to test several interaction schemes. We believe that the present functionality of the SAND Browser would not have been achieved in pure C++, even if an integrated development environment (IDE) was used. We arrive to this conclusion not on the basis of the ease with which user interface components can be rearranged in Tcl/Tk, but rather on the basis of the terseness of the SAND-Tcl interface.

9 Concluding Remarks

Considering the complexity of operations such as spatial joins, an "implementation" such as the one given in Section 5 is surprisingly terse. This is a good example of the intermediate level of abstraction that is possible with a scripting language. While it does not offer the purely declarative abstraction provided by, say, a query language such as SQL, it permits a user or developer to create applications without delving into the intricacies of a full-featured programming language. The design of a set of extensions to be incorporated into a scripting language should be guided by this idea. In other words, the user of the extended language should have at hand all the tools that are necessary to build an application. On the other hand, the programming interface that gives access to these tools should allow the user the maximum liberty in wielding them.

In view of the technical difficulties that are still to be resolved in the field of spatial databases, our experience has been that a scripting language extended with constructs for handling spatial data provides a good trade-off between user-friendliness and richness of features. In particular, SAND-Tcl has enabled the SAND system, while still a research effort, to offer real spatial processing capabilities as well as a modular enough design that will permit it to develop into a more mature environment in the near future.

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] Apple Computer, Inc. *Macintosh Human Interface Guidelines.* Addison-Wesley, Reading, MA, 1992.

[3] L. Becker and R. H. Güting. Rule-based optimization and query processing in an extensible geometric database system. Technical Report 312, Dortmund University, Dortmund, West Germany, August 1989.

[4] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Conference*, pages 322–331, Atlantic City, NJ, June 1990.

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[6] T. Brinkhoff, H. P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proceedings of the ACM SIGMOD Conference*, pages 237–246, Washington, DC, May 1993.

[7] W. Clinger and J. Rees (Editors). Revised(4) report on the algorithmic language Scheme. `http://www-swiss.ai.mit.edu/ftpdir/scheme-reports/r4rs.ps`.

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

[9] M. J. Egenhofer, A. U. Frank, and J. P. Jackson. A topological data model for spatial databases. In A. Buchmann, O. Günther, T. R. Smith, and Y. F. Wang, editors, *Design and Implementation of Large Spatial Databases — First Symposium, SSD'89*, pages 271–286, Santa Barbara, CA, July 1989. (Also Springer-Verlag Lecture Notes in Computer Science 409).

[10] C. Esperança. *Orthogonal objects and their application in spatial databases.* PhD thesis, University of Maryland, December 1995. (Also available as Technical Report TR-3566, University of Maryland, College Park, MD).

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

[12] O. Günther. Efficient computation of spatial joins. In *Ninth International Conference on Data Engineering*, pages 50–59, Kobe, Japan, April 1993. IEEE.

[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, The Netherlands, August 1989.

[14] R. H. Güting. An introduction to spatial database systems. *VLDB Journal*, 3(4):401–444, October 1994.

[15] R. H. Güting, 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, *Advances in Spatial Databases — Fourth International Symposium, SSD'95*, pages 216–239, Portland, ME, August 1995. (Also Springer-Verlag Lecture Notes in Computer Science 951).

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

[17] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In M. J. Egenhofer and J. R. Herring, editors, *Advances in Spatial Databases — Fourth International Symposium, SSD'95*, pages 83–95, Portland, ME, August 1995. (Also Springer-Verlag Lecture Notes in Computer Science 951).

[18] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *Proceedings of the ACM SIGMOD Conference*, Seattle, WA, June 1998.

[19] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, June 1999. (Also University of Maryland Computer Science TR–3919).

[20] E. G. Hoel and H. Samet. Benchmarking spatial join operations with spatial output. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 606–618, Zurich, Switzerland, September 1995.

[21] N. Kabra and D. DeWitt. OPT++ – an object-oriented implementation for extensible database query optimization. See also `http://www.cs.wisc.edu/~ navin/research/apg.html`, 1996.

[22] H. F. Korth and A. Silberschatz. *Database system concepts.* McGraw-Hill, New York, NY, 2nd edition, 1991.

[23] M. Lutz. *Programming Python.* O'Reilly & Associates, Sebastopol, CA, October 1996.

[24] A. Marcus, N. Smilonich, and L. Thompson. *The Cross-GUI Handbook for Multiplatform User Interface Design.* Addison-Wesley, Reading, MA, 1995.

[25] Microsoft Corporation. *The Windows Interface: An Application Design Guide*. Microsoft Press, Redmond, WA, 1992.

[26] S. Morehouse. The architecture of Arc/Info. *Auto-Carto*, 9:266–277, 1989.

[27] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[28] Open Systems Foundation. *OSF/Motif Style Guide*. P T R Prentice Hall, Englewood Cliffs, NJ, 1993.

[29] J. A. Orenstein and F. A. Manola. PROBE spatial data modeling and query processing in an image database application. *IEEE Transactions on Software Engineering*, 14(5):611–629, May 1988.

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

[31] J. Patel, J.B Yu, N. Kabra, K. Tufte, B. Nag, J. Burger, N. Hall, K. Ramasamy, R. Lueder, C. Ellmann, J. Kupsch, S. Guo, J. Larson, D. DeWitt, and J. Naughton. Building a scalable geo-spatial dbms: Technology, implementation, and evaluation. In *Proceedings of the ACM SIGMOD Conference*, pages 336–347, AZ, 1997.

[32] S. Prabhakar, D. Agrawal, A. El Abbadi, A. Singh, and T. Smith. Browsing and placement of multiresolution images on parallel disks. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, pages 102–113, San Jose, CA, November 1997.

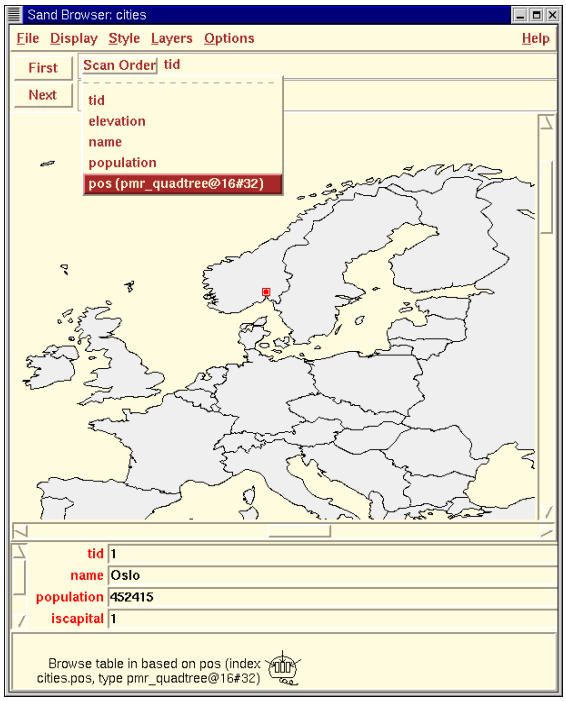
[33] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[34] M. Stonebraker. Limitations of spatial simulators for relational DBMSs. Technical report, INFORMIX Software, Inc., 1997. (see `http://www.informix.com/informix/corpinfo/zines/whitpprs/wpsplsim.pdf`).

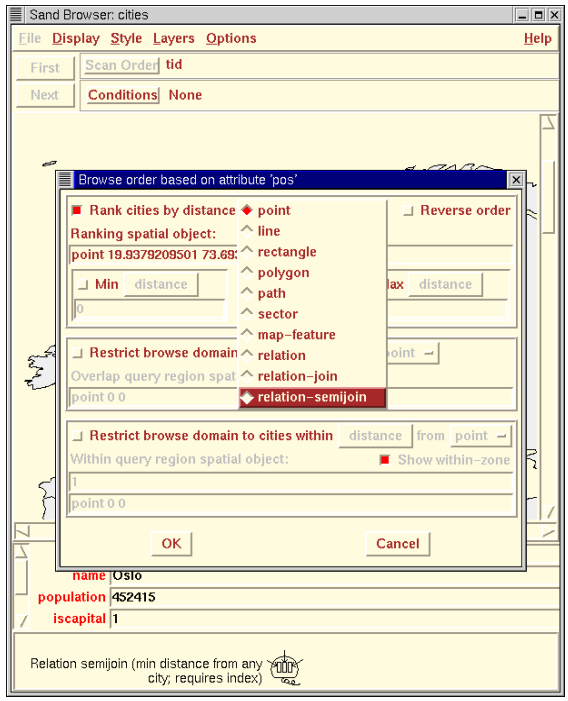
[35] M. Stonebraker, T. Sellis, and E. Hanson. An analysis of rule indexing implementations in data base systems. In *Proceedings of the First International Conference on Expert Database Systems*, pages 353–364, Charleston, SC, April 1986.

[36] T. Vijlbrief and P. van Oosterom. The GEO++ system: An extensible GIS. In *Proceedings of the 5th International Symposium on Spatial Data Handling*, pages 40–50, Charleston, SC, August 1992.

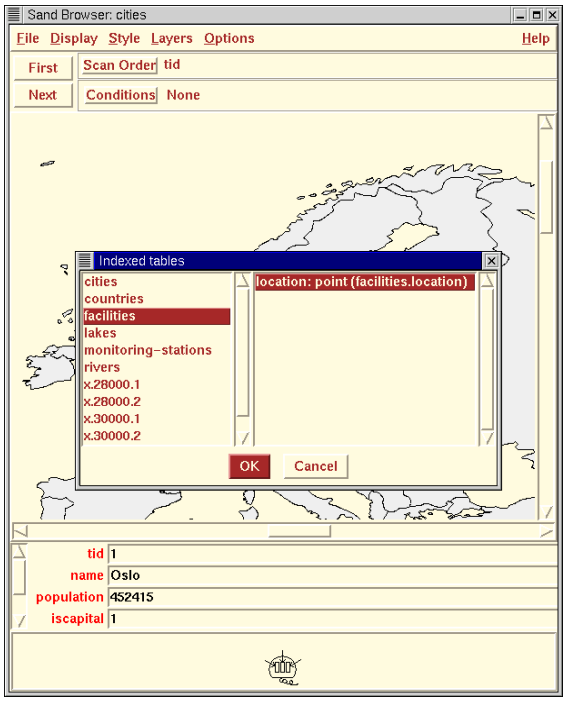
[37] A. Voisard. Towards a toolbox for geographic user interfaces. In O. Günther and H. J. Schek, editors, *Advances in Spatial Databases — Second Symposium, SSD'91*, pages 75–97, Zurich, Switzerland, August 1991. (Also Springer-Verlag Lecture Notes in Computer Science 525).



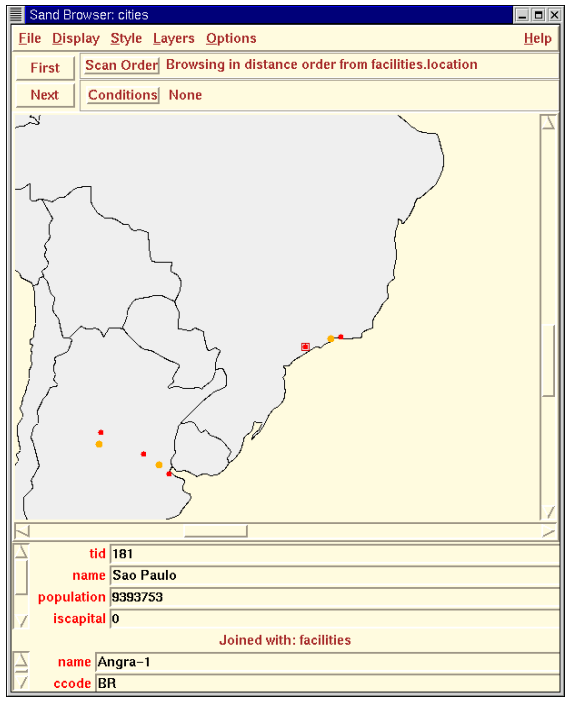
(a)



(b)



(c)



(d)

Figure 12: Sand Browser interface. (a) User requests to browse the *city* relation based on the attribute *pos*; (b) selects distance semi-join query from the dialog box; and (c) selects the *facility* relation corresponding to the nuclear facilities. Successive city/nuclear facility pairs, denoted by dark/light dots in (d), are displayed each time the "Next" button is pressed.