The Spatial Spreadsheet *

Glenn S. Iwerks
Computer Science Department,
University of Maryland, College Park, Maryland 20742
iwerks@umiacs.umd.edu

Hanan Samet †
Computer Science Department, Institute for Advanced Computer Studies
University of Maryland, College Park, Maryland 20742
hjs@cs.umd.edu

Abstract

The power of the spreadsheet can be combined with that of the spatial database to provide a system that is flexible, powerful and easy to use. In this paper we propose the Spatial Spreadsheet as a means to organize large amounts of spatial data, to quickly formulate queries on that data, and to propagate changes in the source data to query results on a large scale. Such a system can be used to organize related queries that not only convey the results of individual queries but also serve as a means of visual comparison of query results.

Keywords: spreadsheets, spatial databases, visualization

1 Introduction

In this paper we introduce the Spatial Spreadsheet. The purpose of the Spatial Spreadsheet is to combine the power of a spatial database with that of the spreadsheet. The advantages of a spreadsheet is the ability to organize data, to formulate operations on that data quickly through the use of row and column operations, and to propagate changes in the data through the system.

The Spatial Spreadsheet consists of a 2D array of cells containing data. Updates can propagate through the array via cell operations. Operations can be single cell operations, row operations, or column operations. Column operations iterate over rows in a column and row operations iterate over columns in a row. Cell values can be instantiated by the user or can be a result of operations performed on other cells. In the classic spreadsheet paradigm, cell values are primitive data types such as numbers and strings whereas in the Spatial Spreadsheet, cells access database relations. The relation is part of a spatial relational database. A relation is a table of related attributes. A tuple in a relation is one instance of these related items. Each table is made up of a set of tuples [9]. Attributes in a spatial database relation can be primitive types such as numbers and strings or spatial data types such as points, lines and polygons. Spatial attributes stored in the relations associated with each cell can be displayed graphically for visualization of query results. This allows the effects of updates on the base input relations to be observed through the graphical display when changes occur.

*VISUAL'99, *pages 317-324, Amsterdam, The Netherlands, June 1999*
†The support of the National Science Foundation under Grant IRI-97-12715 is gratefully acknowledged.

The rest of this paper is organized as follows. Section 2 gives some background on spreadsheets and spatial databases. Section 3 describes the Spatial Spreadsheet. Section 4 provides some implementation details. Section 5 draws some concluding remarks as well as gives some directions for future research.

2 Background

2.1 The Classic Spreadsheet

The classic spreadsheet was designed as an accounting tool. It permitted the user to quickly formulate calculations on the data through column and row operations. It also allowed the user to easily observe how changes in the input data affected a whole series of calculations. The original spreadsheet was laid out in a two-dimensional array of cells in rows and columns. Users could populate the rows and columns with numeric data. They could then perform operations on entire columns (or rows) and populate additional columns with the results.

2.2 Spreadsheet for Images

Spreadsheets for Images (SI) is an application of the concept of a spreadsheet to the image processing domain [6]. In this case, the concept of a spreadsheet is used as a means of data visualization. Each cell in the spreadsheet contains graphical objects such as images and movies. Formulas for processing data can be assigned to cells. These formulas can use the contents of other cells as inputs. This ties the processing of data in the cells together. When a cell is modified, other cells that use it as input are updated. A somewhat related capability is provided by the CANTATA programming language to be used with the KHOROS system [8].

2.3 SAND Browser

The SAND Browser is a front end for the SAND [2] spatial relational database. The user need only to point and click on a map image to input spatial data used in the processing of query primitives. The results of the queries are then displayed graphically. This gives the user an intuitive interface to the database to help the visualization of the data and the derivation of additional information from it. However, such a system does have limitations. In the SAND Browser one primitive operation is processed at a time. When the user wants to make a new query, the results of the previous operation are lost unless they are saved explicitly in a new relation. As a result, there is no simple and implicit way to generate more complicated queries from the primitives. In presenting the Spatial Spreadsheet we will propose some possible solutions to these limitations of the SAND Browser while still maintaining its ease of use and intuitive nature.

3 The Spatial Spreadsheet

The Spatial Spreadsheet is a front end to a spatial database. A spatial database is a database in which spatial attributes can be stored. Attributes of a spatial relational database may correspond to spatial and non-spatial data. For example, spatial data types may consist of points, lines, and polygons. Numbers and character strings are examples of non-spatial data. By mapping the world coordinates of the spatial data to a bitmap it may be converted to an image for visualization of the data. The Spatial Spreadsheet provides a means to organize the relational data and query results in a manner that is intuitively meaningful to the user. One may apply meaning to a column, a row, or an entire set of columns or rows to organize data. For example, spatio-temporal data may be

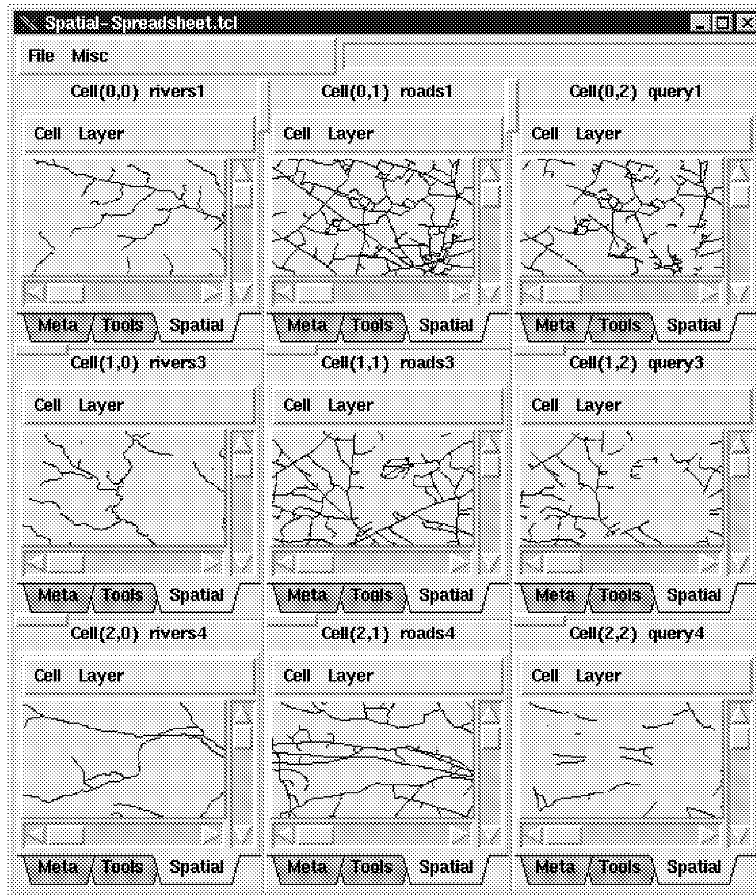

Figure 1: Example query results in top-level window

organized so that each row corresponds to a different time period and each column corresponds to a different region in the world.

The Spatial Spreadsheet is made up of a 2D array of cells. Each cell in the spreadsheet can be referenced by the cell's location (row, column). In the Spatial Spreadsheet, each cell represents a relation. A cell can contain two types of relations: a persistent relation or a query result. A persistent relation is a relation that exists in a permanent state. This is not to say that the data in the relation does not change but rather that the relation existed before the spreadsheet was invoked and will continue to exist after the spreadsheet exits unless explicitly deleted by the user. The second type of a relation contains the result of a query posed by the user. The user decides if a query result will persist or not. The user can pose simple queries. Simple queries are primitive operations. Some examples of a primitive operation are selection, projection, join, spatial join [5], window [1], nearest neighbor [4], etc. Primitive operations are composed to create complex queries.

3.1 Example

Let us consider a simple example (see Figure 1). Suppose that we are concerned about flooding in 3 different regions of the world: A, B and C. Roads close to rivers may get washed out when the rivers flood. We want to know what roads in these regions are close to a river at or near flood stage.

For each of these regions we have a relation containing all the rivers at or near flood stage. We open these river relations in the first column of our spreadsheet (i.e., column 0). We let row 0 correspond to region A, row 1 to region B, and row 2 to region C. We open relations in column 1 that store position information for roads in each region. Our column operation is to find all the roads in cells in column 1 that are within 500 meters of a river in the cell in column 0 of the same row and store the result in column 2. In a modified version of SQL [9] the query might look as follows.

```
SELECT *
FROM Cell(X,1), Cell(X,2),
    distance(Cell(X,1).river, Cell(X,2).road) d
WHERE d < 500
```

The modification to SQL[3] introduced here is the Cell() function. Instead of giving an explicit relation name in the FROM clause, we introduce the Cell() function that takes a row and a column value and returns a relation. The presence of the variable X for the row parameter tells the system to iterate over all open relations in the given columns.

The operation producing the result in column 3 is an example of a column operation. Similarly, one can iterate over all the columns in a row using a row operation. One can also perform single cell operations.

3.2 Design

The design of the Spatial Spreadsheet is object-oriented. Figure 2 shows the basic object model of the Spatial Spreadsheet in UML notation [3]. The figure shows six class objects: *Spreadsheet*, *Cell*, *Display*, *Relation*, *Query* and *Processor*. It is important to note the distinction between a *Cell* object and what has been previously referred to as a cell. A cell is an element in the spreadsheet array. A *Cell* object is a class object named "*Cell*" used in the design and underlying implementation of the spreadsheet. Likewise, a *Relation* object is the class object named "*Relation*" not to be confused with a relation in the relational database. In the remainder of this paper we will distinguish object names by using the italic font.

When the Spatial Spreadsheet is started, an instance of the *Spreadsheet* object is created. This is the top-level object and acts as the root aggregator to all other objects. The primary responsibility of the *Spreadsheet* object is to keep track of *Cell* objects, global states, and the organization of cells in the top-level window of the graphical user interface. A *Spreadsheet* object can have one or more *Cell* objects. *Query* objects and *Relation* objects are *Cell* objects — that is, they are derived from *Cell* objects. An instance of a *Cell* object is created when a persistent relation is opened or a cell is needed to process and store a primitive operation. *Cell* objects have member data items to keep track and manipulate their own relation. *Cell* objects can be associated with other *Cell* objects. *Query* objects derived from *Cell* objects use these associations to keep track of which other *Cell* objects it uses as input. All *Cell* objects use these associations to keep track of which *Query* objects use them as input. This becomes important in update propagation. Each *Cell* object has a *Display* object. The *Display* object's role is to display data from the relation for the user. *Display* objects can display information for the user in several ways including a meta data display, tuple-by-tuple display of raw data, and a graphical display for spatial data types. In the graphical display spatial attributes are rendered by projecting their coordinates onto a 2D bitmap as a means of data visualization. Each *Query* object also has a *Processor* object. *Processor* objects are responsible for processing primitive operations.

[3]SQL is not actually used in the Spatial Spreadsheet system. It is only used here for example purposes.

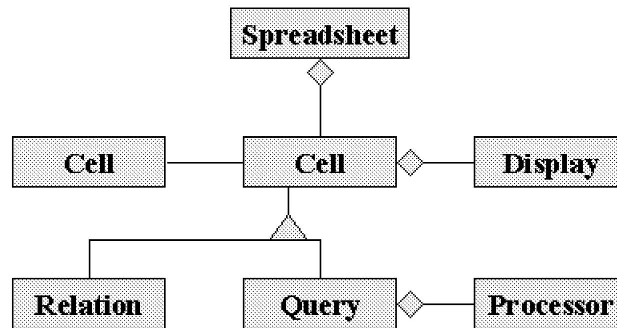

Figure 2: Spatial Spreadsheet Object Model: boxes indicate class objects, diamonds indicate aggregate or "has a" relationships, and triangles indicate inheritance.

3.3 Update Propagation

There are two ways the data stored in a relation open in the spreadsheet can be changed. The first way is by an outside source. In particular, another process that accesses the underlying database can make changes to the data. The second way is by the actions of the spreadsheet itself. If a persistent relation is updated by an outside source, the effects of those changes need to be propagated to all the other cells that directly or indirectly use that relation as input. Consider the river and road example. Suppose it has been raining a lot in region B and the relation containing the information on rivers at or near flood stage is updated by inserting more rivers. In this case, the *Cell* object holding the result in column 2 for region B would need to be updated after the change occurred in column 0.

The propagation process works as follows. A relation corresponding to a *Relation* object is updated. The *Relation* object is notified and it marks itself as "dirty". When a *Relation* object or a *Query* object becomes dirty it then informs all *Cell* objects depending on it for input that they are now dirty too. It may be useful to think of the *Cell* objects in the spreadsheet as nodes in a directed graph. Edges directed into a node indicate *Cell* object inputs. Nodes in the graph having no incoming edges are *Relation* objects. All the other nodes are *Query* objects. We will refer to *Query* objects that have no outgoing edges as terminals. The manner in which queries are created ensures that there are no cycles in this directed graph. Therefore, we do not have to check for cycles while passing messages. Eventually, these messages are passed through all possible paths from the initial dirty *Relation* object to all terminals reachable from the initial *Relation* object. Since there are no cycles, message passing will cease.

After all *Cell* objects are marked dirty that can be marked dirty, the initial dirty *Relation* object marks itself as "clean". The PropagateClean() method is invoked for each *Cell* object that uses the *Relation* object as direct input. The PropagateClean() method propagates the update.

```
PropagateClean() {
  If all my inputs are clean and I am active then {
    Mark myself clean and recalculate primitive operation
    For each Cell object J that uses me as input do
      Call J's PropagateClean() method
  }
}
```

It is necessary to propagate all the "dirty" messages all the way through the graph of *Cell* objects before recalculating any primitive operations associated with a *Cell* object otherwise some *Cell* objects might recalculate their operations more than once. For example, suppose that *Cell* object X recalculates its operation as soon as one of its inputs, say *Cell* object Y, indicates that a change has occurred. If *Cell* object Y is also input to *Cell* object Z which in turn is input to *Cell* object X, then *Cell* object X would have to update itself again after it is informed that *Cell* object Z has been updated. If this situation was not prevented, then there could be as many as $O\{n^2\}$ updates. This situation is prevented by informing each *Cell* object of all imminent updates before any updates are actually performed. This essures $O\{n\}$ updates.

Note that individual *Cell* objects may be set "active" or "inactive" by the user. An inactive *Cell* object blocks the update of itself and blocks the propagation of updates to it's dependents. This avoids spending time updating *Cell* objects in the spreadsheet in which the user is not currently interested.

Updates may propagate automatically whenever a change occurs or only as desired by the user. At the top level, the *Spreadsheet* object has an UpdateSpreadsheet() method. This is called to initiate update propagation.

```
UpdateSpreadsheet() {
  For each Cell object K in the spreadsheet do
    If K is dirty then
      Call K's PropagateClean() method.
}
```

3.4 Graphical User Interface

Rather than expressing operations on cells with a query language such as SQL, the simple operations associated with cells are created through the use of the "wizard". The wizard consists of one or more popup windows that guide the user through the steps of instantiating a cell. To start the wizard, the user clicks on an empty cell. At each step, the wizard offers the possible choices to the user and the user selects the desired choice with the mouse. In some cases, the user may still have to type something. In particular, this is the case when an expression is required for a selection or join operation. At present, the user is required to type the entire expression. As in the SI system [6], we chose to use Tcl [7] for expressions. This requires the user to be knowledgeable of the expression syntax. This error-prone aspect detracts from the GUI's ease of use. We intend to replace this with a more intuitive system in the future.

The main window consists of an array of cells (see Figure 1). Cells can be expanded or contracted by sliding the row and column boundaries back and forth. Theoretically, the spreadsheet could hold an unlimited number of rows and columns but to simplify the implementation we limit the number of rows and columns. We can still start the system with a large number of cells and hide those that are not being used by moving the sliders.

Display of spatial attributes is not limited to the graphical display in a single cell. Each graphical display can display spatial attributes from any relation associated with any cell in the spreadsheet. This allows the user to make visual comparisons by overlaying different layers in the display. The Spatial Spreadsheet also provides a global graphical display in a separate top-level window.

4 Implementation

The Spatial Spreadsheet is an interface used to interact with a spatial relational database. The spatial relational database we use is SAND [2]. SAND provides the database engine that underlies the system. It contains facilities to create, update and delete relations. It provides access methods and primitive operations on spatial and non-spatial data. The Spatial Spreadsheet extends the basic set of primitive queries to include the classic selection, projection and nested loop join operations. The implementation of the Spatial Spreadsheet is object-oriented and was written entirely in incremental Tcl (iTcl) and incremental Tk (iTk). It runs on Sun Sparc and Linux systems.

5 Concluding Remarks

We have described how the power of the spreadsheet can be combined with a spatial database. The Spatial Spreadsheet provides a framework in which to organize data and build queries. Row and column operations provide a mechanism for rapid query creation on large amounts of related data. The systematic tabulation of the data as found in the two-dimensional array of the Spatial Spreadsheet enables the user to visually compare spatial components and pick out patterns. The user can also see how query results change as updates occur. An important issue for future work that was not addressed here is update propagation optimization. In particular, the output of any given *Query* object may be the result of many steps along the way between it and initial *Relation* objects. Currently the method of computation is determined in a procedural manner by the user. In the future we will focus on converting this to a declarative form and using query optimization techniques to improve refresh efficiency when updates occur.

References

[1] W. G. Aref and H. Samet. Efficient window block retrieval in quadtree-based spatial databases. *GeoInformatica*, 1(1):59–91, April 1997.

[2] C. Esperança and H. Samet. Spatial database programming using SAND. In M. J. Kraak and M. Molenaar, editors, *Proceedings of the Seventh International Symposium on Spatial Data Handling*, volume 2, pages A29–A42, Delft, The Netherlands, August 1996.

[3] M. Fowler and K. Scott. *UML Distilled, Applying the Standard Object Modeling Lanuage.* Addison-Wesley, Reading, MA, 1997.

[4] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. Computer Science Department TR-3919, University of Maryland, College Park, MD, July 1998. (To appear in *ACM Transactions on Database Systems*).

[5] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *Proceedings of the ACM SIGMOD Conference*, pages 237–248, Seattle, WA, June 1998.

[6] M. Levoy. Spreadsheets for images. In *Proceedings of the SIGGRAPH'94 Conference*, pages 139–146, Los Angeles, 1994.

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

[8] J. Rasure and C. Williams. An integrated visual language and software development environment. *Journal of Visual Languages and Computing*, 2(3):217–246, September 1991.

[9] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts.* McGraw-Hill, New York, third edition, 1996.