SaIL: A Library for Efficient Application Integration of Spatial Indices

Marios Hadjieleftheriou*, Erik Hoel†, Vassilis J. Tsotras*
* Computer Science Department
University of California, Riverside
Email: marioh, tsotras@cs.ucr.edu
† Environmental Systems Research Institute
380 New York Street
Redlands, CA 92373
Email: ehoel@esri.com

***Abstract*— Many scientific applications deal with spatial, spatiotemporal and other multidimensional indexing structures, typically managing millions of objects with arbitrary and complex features. Choosing the appropriate method to index such data becomes rather difficult. Having an index library that can combine different indices under the same programming interface is thus very valuable. In this paper we present *SaIL* (SpAtial Index Library), a robust and extensible library that enables simple integration of spatial index structures in existing applications. We mainly focus on design issues and elaborate on techniques for making the framework generic enough, so that it can support user defined data types, customizable spatial queries, and a broad range of spatial (and spatio-temporal) index structures. The library is publicly available and has already been successfully utilized for research and commercial applications.**

I. Introduction

It is well recognized that a plethora of scientific and other applications deal with spatial, spatiotemporal and generally multidimensional data. Typically, such applications manage millions of objects with arbitrary and complex spatial features. Examples include GIS applications that manage maps with numerous layers and hundreds of thousands of features [1], astronomical applications like the SkyServer [2] indexing millions of images, traffic analysis and surveillance applications that track numerous moving objects, and bioinformatics applications about thousands of genes.

Usually, the end-user of such applications is interested in analyzing a small fragment of the data at a time, issuing various advanced spatial queries. The utility of spatial indexing techniques for such applications has been well recognized; complex queries can be answered efficiently only with the use of such structures (e.g., nearest neighbor and top-k queries). Consequently, many indexing techniques aiming at solving disparate problems have appeared lately in the literature. Every technique has its own advantages and disadvantages, being suitable for different application domains or dataset types. Therefore, choosing an appropriate access method for the problem at hand is rather difficult. Hence, it becomes evident that a spatial index library which can combine all index structures under a common application programming interface would be very valuable to the user.

The major difficulty of such an undertaking is that most index structures have diverse characteristics that distinguish them. For example they may employ data or space partitioning, they could have rectangular, spherical or other node types, they are balanced or unbalanced, they index points, rectangles, lines or other shapes, they can have arbitrary dimensionality, etc. In addition, programmers should be able to use access methods that can exploit the semantics of application-specific data types by customizing existing structures, while making sure that meaningful queries can be formulated easily for the specific data types.

Moreover, it is vital to adopt a common design framework in order to promote reusability and familiarity, especially for large applications where many developers are involved. The framework should capture the most important design characteristics, common to most structures, into a concise set of interfaces so that developers can concentrate on other aspects of the development process. Its interface should be easily extensible to address future needs without necessitating revisions of client code.

These fundamental requirements make the design of a *generic* spatial index framework a challenging task. Even though there is a substantial volume of work on spatial index structures and their properties, little work has appeared that addresses design and implementation issues. Towards this aim, this paper presents *SaIL*, a spatial index library that enables simple integration of spatial and spatio-temporal index structures into existing applications. A sample implementation is publicly available (http://www.cs.ucr.edu/∼marioh/spatialindex) and has been used successfully for both research and commercial purposes (e.g., UCR and ESRI). We proceed with a discussion of related work, followed by a high-level description of SaIL.

II. Related Work

The most relevant work to ours is XXL [5]. The eXtensible and fleXible Library offers both low-level and high-level components for development and integration of spatial index

This work was partially supported by NSF grants IIS-9907477, EIA-9983445, IIS-0220148.

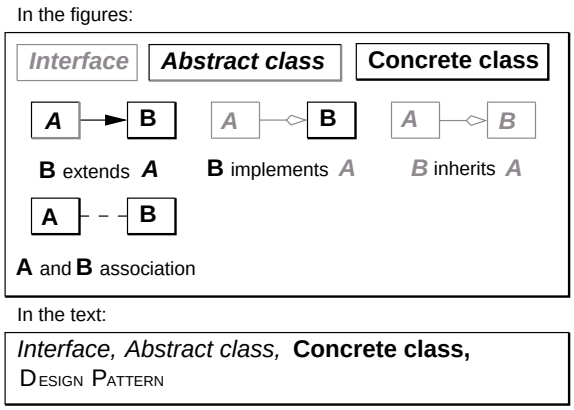

Fig. 1. Notation used in the paper.

structures. Even though XXL is a superset of SaIL, it differs in two major aspects. First, our implementation offers a very concise, straightforward interface for querying different index structures in a uniform manner. In contrast, XXL querying interfaces are index specific. Second, we offer a more generalized querying capability. Despite the fact that XXL can support a variety of advanced spatial queries, user defined queries have to be implemented by hand requiring modifications in all affected index structures. In contrast, we offer an intuitive interface for formulating novel queries without having to revise the library in any way.

GiST (for *Generalized Search Tree* [8]) is also relevant to our work. GiST is a framework that generalizes a height balanced, single rooted search tree with variable fanout. By using a simple interface, GiST can support a wide variety of search trees and their corresponding querying capabilities. Various papers have been recently proposed to make GiST more generic [3], [4].

Our work is orthogonal to GiST and its variants. GiST and its extensions address the implementation issues behind new access methods by removing the burden of writing structural maintenance code from the developer. SaIL does not aim at simplifying the development process of the index structures themselves, but more importantly, the development of the applications that use them.

III. SPATIAL INDEX LIBRARY ARCHITECTURE

In this section we present the Spatial Index Library in more detail. We analyze the most important concepts behind each design decision and give useful examples. Figure 1 summarizes the notation used in the text and diagrams. When referring to specific design patterns we use the definitions in Gamma et al. [6].

A. The Core Toolkit

This toolkit addresses very simple but essential needs for any generic framework. It provides a **Variant** type for representing a variety of different primitive types (like integers, floats, character arrays, etc.), which is necessary for avoiding hard coding specific primitive types in interface definitions that might need to be modified at a later time. It offers a **PropertySet**, or a collection of $< PropertyName, Value >$ pairs. Property sets are useful for passing an indeterminate number of parameters to a method, even after the interfaces have been defined, without the need to extend them. It also provides an *Exception* class hierarchy for promoting the use of exception handling in client code, since failure of index structure components is not a rare situation. Finally, it provides other utility classes like external sorters, comparators, etc., details of which are omitted due to lack of space.

B. The Storage Manager Toolkit

An essential, critical part of spatial indexing tools is the storage manager. It should be versatile, very efficient and provide loose coupling. Clients that want to persist entities should be unaware of the underlying mechanisms, in order to achieve proper encapsulation. Persistence could be over the network, on a disk drive, in a relational table, etc. All mediums should be treated uniformly in client code, in order to promote flexibility and facilitate the improvement of storage management services as the system evolves.

The storage manager toolkit is shown in Figure 2. The key abstraction is a MEMENTO pattern that allows loose coupling between the objects that are persisted and the concrete implementation of the actual storage manager. An object that wants to store itself has to instantiate a concrete subclass of *Memento* that accurately represents its state. Then, it can pass this instance to a component supporting the *IStorageManager* interface, which will in turn return an identifier that can be used to retrieve the object's state at a later time.

The *IBuffer* interface provides basic buffering capabilities. Using the classes that implement *IBuffer* is straightforward; they act as a proxies between a storage manager and the client that uses it, buffering entries as they see fit. Conveniently, the client is unaware that buffering is taking place by assuming that it is interfacing with a storage manager directly. This architecture provides sufficient flexibility to even alter buffering policies at runtime.

C. The Spatial Index Interface

Spatial access methods index complex spatial objects with varying shapes. In order to make our interfaces generic it is essential to have a basic shape abstraction that can also represent composite shapes and other decorations (meta-data like z-ordering, insertion time, etc.). We define the *IShape* COMPOSITE pattern (Figure 3) as an interface that all index structures should use to decouple their implementation from actual concrete shapes. For example, inserting convex polygons into an R-tree [7] can be accomplished by calling the *IShape*.$getMBR$ method to obtain the minimum bounding region of the polygon. The R-tree can remain unaware of the details of convex polygon representations. Complex shapes can be represented by composing different *IShapes* under one class. Hence, they can be handled uniformly.

Another important capability of a generic framework is to provide a sound set of index elements (leaf and index nodes, data elements, etc.) that enable consistent manipulation of diverse access methods. For example, querying functions

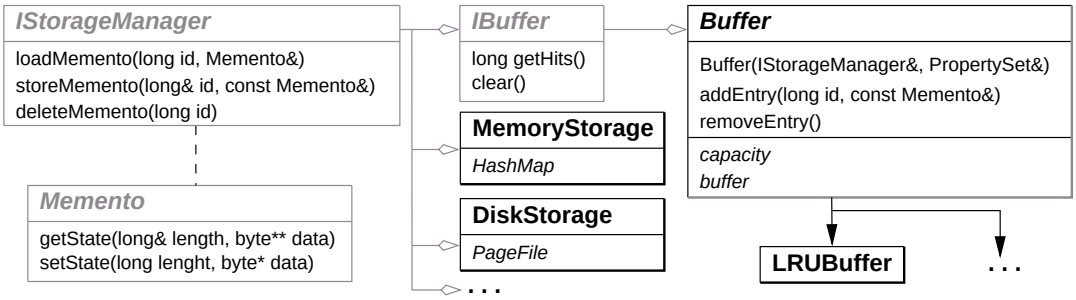

Fig. 2. The Storage Manager Toolkit.

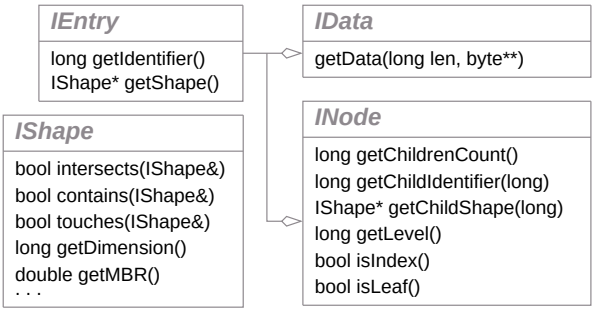

Fig. 3. The IShape and Spatial Index Elements Interfaces.

should return iterators (i.e., enumerations or cursors) over well-defined data elements, irrespective of what kind of structures they operate on. We achieve this by providing the following hierarchy: *IEntry* is the most basic interface for a spatial index entry; its members are an identifier and a shape. *INode* (that inherits *IEntry*) represents a generic tree node; its members are the number of children it contains, its tree level, and if it is an index or a leaf. The *IData* interface represents a data element; it contains the meta-data associated with the entry or a pointer to the real data.

The core of the spatial index interface is the *ISpatialIndex* FACADE pattern. All index structures should implement *ISpatialIndex* (apart from their own custom methods), which abstracts the most common index operations. This interface is as generic as possible.

The $insertData$ method accepts the data object to be inserted as an *IShape*, an interface that can be used as a simple decorator over the actual object implementation. Meta-data can also be stored along with the object as byte arrays. The $deleteData$ method is straightforward. It accepts the *IShape* to be deleted and its object identifier.

The query methods take a query *IShape* as an argument. This simple interface is powerful enough to allow the developer to create customized queries (this is one of the main differences with XXL). For example, suppose a circular range query on an R-tree is required. Internally, the R-tree search algorithm decides if a node should be examined by calling the query $intersects$ predicate on a node's MBR. Hence, it suffices to define a **Circle** class that implements the $intersects$ function (specific for intersections between circles and MBRs) and call the $intersectionQuery$ method with a **Circle** object as its argument. Since arbitrarily complex shapes can be defined with the *IShape* interface, the querying capabilities of the index structures are only limited by the ability of the developer to implement correctly the appropriate predicate functions.

To be able to customize querying behavior even further a VISITOR pattern is used. The *IVisitor* interface is a very powerful feature (something that XXL does not provide). The query caller can implement an appropriate visitor that executes user defined operations when index entries are accessed. For example, the visitor can ignore all node entries and cache all visited data entries (essentially the answers to the query) for later processing (like an enumeration). Instead, it could process the answers interactively (like a cursor), terminating the search when desired. Other useful examples are tallying the number of query I/Os and visualizing the query progress. (A visitor example is presented in the Appendix.)

The *IShape* and *IVisitor* interfaces enable consistent and straightforward query integration into client code, increasing readability and extensibility. New index structures can add specialized functionality by requesting decorated *IShape* objects (thus, without affecting the interfaces). The *IVisitor* interface allows existing visitor implementations to be reused for querying different types of access methods and users can customize visitors during runtime.

The nearest neighbor query method can be customized even further. Since different applications use different distance measures to find nearest neighbors, it accepts an *INearestNeighborComparator* object. By allowing the caller to provide a customized comparator, the default nearest neighbor algorithm implemented by the underlying structure can be used, obviating any changes.

For implementing "exotic" queries, without the need to make internal modifications to the library, a STRATEGY pattern is proposed (another advanced feature that XXL is lacking). Using the $queryStrategy$ method the caller can fully guide the traversal order and the operations performed on a structure's basic elements allowing, in effect, the construction of custom querying algorithms on the fly. This technique uses an *IQueryStrategy* object for encapsulating the traversal algorithm. The index structure calls *IQueryStrategy*.$getNextEntry$ by starting the traversal from a root and the IQueryStrategy object chooses which entry should be accessed and returned next. The traversal can be terminated when desired. As an example, assume that the user wants to visualize all the index levels of an R-tree. Either the

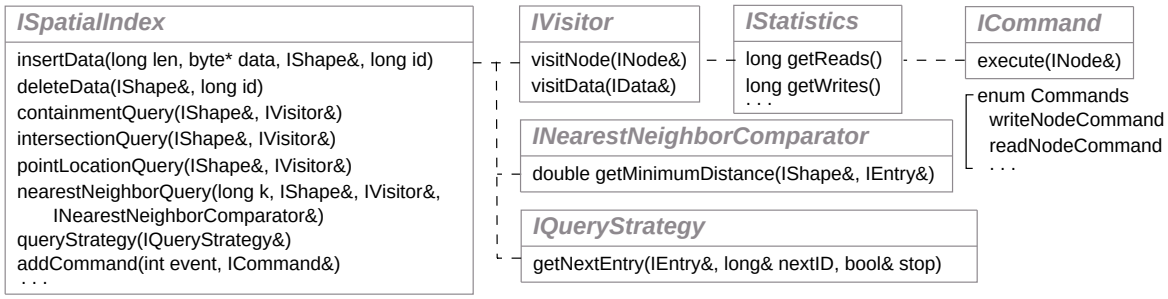

Fig. 4. The Spatial Index Interface.

R-tree implementation should provide a custom tree traversal method that returns all nodes one by one, or a query strategy can be defined for the same purpose (which can actually be reused as is, or maybe with slight modifications, for any other hierarchical structure). An example of a breadth-first node traversal algorithm is presented in the Appendix (the example requires less than 15 lines of code). The possible uses of the query strategy pattern are boundless.

Another capability that should be provided by most index structures is allowing users to customize various index operations (usually by the use of call-back functions). The spatial index interface uses a COMMAND pattern for that purpose. It declares the *ICommand* interface; objects implementing *ICommand* encapsulate user parametrized requests that can be run on specific events, like customized alerts. All access methods should provide a number of queues, each one corresponding to different events that trigger each request. For example, assume that we are implementing a new index structure. We can augment the function that persists a node to storage with an empty list of *ICommand* objects. Using the $addCommand$ method the user can add arbitrary command objects to this list, that get executed whenever this function is called, by specifying an appropriate event number (an enumeration is provided for that purpose). Every time the function is called, it iterates through the *ICommand* objects in the list and calls their execute method. The COMMAND pattern promotes reusability, clarity, and ease of extensibility without the need of subclassing or modifying the spatial index implementations simply to customize a few internal operations as dictated by user needs.

IV. CONCLUSIONS

We presented a robust and extensible spatial index developer's framework. We argue that the proposed framework will help developers incorporate spatial access methods with greater ease into existing applications. The framework is based on well documented design patterns that promote reusability and improved code quality. The demo will include various examples (like the Visitor and Query patterns in the Appendix) as well as examples on how the basic interface can be used to integrated SaIL in an application. A sample implementation in C++ and Java can be downloaded from http://www.cs.ucr.edu/∼marioh/spatialindex.

REFERENCES

[1] ArcGIS. http://www.esri.com/software/arcgis/index.html.

[2] SkyServer. http://skyserver.sdss.org/dr1/en/.

[3] P. M. Aoki. Generalizing "search" in generalized search trees (extended abstract). In *Proc. of International Conference on Data Engineering*, pages 380–389, 1998.

[4] W. G. Aref and I. F. Ilyas. An extensible index for spatial databases. In *Proc. of Scientific and Statistical Database Management*, pages 49–58, 2001.

[5] J. Van den Bercken, B. Blohsfeld, J. Dittrich, J. Krämer, T. Schäfer, M. Schneider, and B. Seeger. XXL - a library approach to supporting efficient implementations of advanced database queries. In *Proc. of Very Large Data Bases*, pages 39–48, 2001.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[7] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of ACM Management of Data*, pages 47–57, 1984.

[8] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proc. of Very Large Data Bases*, pages 562–573, 1995.

APPENDIX

A. Design Pattern Descriptions

MEMENTO: Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

PROXY: Provide a surrogate or place holder for another object to control access to it.

COMPOSITE: Composite lets clients treat individual objects and compositions of objects uniformly.

FACADE: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

VISITOR: Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

STRATEGY: Define a family of algorithms, encapsulate each one, and make them interchangeable.

COMMAND: Encapsulate a request as an object, thereby letting you parameterize clients with different requests.

B. Visitor and Query Strategy Examples

TABLE I
IVisitor EXAMPLE.

```
class MyVisitor : public IVisitor {
public:
 map<long, IShape*> answers;
 long nodeAccesses;

 MyVisitor() : nodeAccesses(0) {}

 public visitNode(INode* n) {
  nodeAccesses++;
 }

 public visitData(IData* d) {
  // add the answer to the list.
  answers[d.getIdentifier()] = d.getShape();
 }
};
```

TABLE II
IQueryStrategy EXAMPLE.

```
class MyQueryStrategy
      : public IQueryStrategy {
 queue<long> ids;
public:
 void getNextEntry(IEntry& e,
          long& nextID, bool& stop) {
  // process the entry.
  . . .

  // if it is an index entry and not a leaf
  // add its children to the queue.
  INode* n = dynamic_cast<INode*>(&e);
  if (n != 0 && ! n->isLeaf)
   for (long cChild = 0;
        cChild < n->getChildrenCount();
        cChild++)
    ids.push(n->getChildIdentifier(cChild));

  stop = true;
  if (! ids.empty()) {
   // if queue not empty fetch the next entry.
   nextID = ids.front(); ids.pop();
   stop = false;
  }
 }
};
```