SaIL: A Spatial Index Library for Efficient Application Integration

Marios Hadjieleftheriou
Computer Science Department
Boston University
111 Cummington St., Boston, MA
marioh@cs.bu.edu

Erik Hoel
Research and Development
Environmental Systems Research Institute
380 New York St., Redlands, CA 92373
ehoel@esri.com

Vassilis J. Tsotras
Computer Science Department
University of California, Riverside
Riverside, CA 92521
tsotras@cs.ucr.edu

Abstract

With the proliferation of spatial and spatio-temporal data that are produced everyday by a wide range of applications, Geographic Information Systems (GIS) have to cope with millions of objects with diverse spatial characteristics. Clearly, under these circumstances, substantial performance speed up can be achieved with the use of spatial, spatio-temporal and other multi-dimensional indexing techniques. Due to the increasing research effort on developing new indexing methods, the number of available alternatives is becoming overwhelming, making the task of selecting the most appropriate method for indexing the data according to application needs rather challenging. Therefore, developing a library that can combine a variety of indexing techniques under a common application programming interface can prove to be a valuable tool. In this paper we present *SaIL* (SpAtial Index Library), an extensible framework that enables easy integration of spatial and spatio-temporal index structures into existing applications. We 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, in a way that does not compromise functionality, extensibility and, primarily, ease of use. SaIL is publicly available and has already been successfully utilized for research and commercial applications.

1 Introduction

There is a plethora of GIS and related applications that are related with spatial, spatio-temporal and, generally, multi-dimensional data. Typically, such applications manage millions of objects with diverse spatial characteristics. Examples include mapping applications that have to visualize numerous layers and hundreds of thousands of features [13], astronomical applications that

This work was conducted while the first author was visiting ESRI and was partially supported by ESRI, NSF grants IIS-9907477, EIA-9983445, and IIS-0220148.

index millions of images [26], traffic analysis and surveillance applications that track thousands of vehicles, and more.

Usually, the end-user of these applications is interested in analyzing a small fragment of the data at a time, while issuing various queries, spatial or more generic in nature. Nevertheless, the utility of spatial indexing techniques for such applications has been well recognized — complex spatial queries can be answered efficiently only with the use of spatial index structures (for instance, nearest neighbor queries). Consequently, many indexing techniques aiming at solving disparate problems and optimized for diverse types of spatial queries have appeared lately in the literature (like the R-tree [18], the X-tree [6], etc.; excellent surveys on spatial index structures appear in [16, 7]). As a result, each technique has specific advantages and disadvantages that make it suitable for different application domains and dataset types. Therefore, the task of selecting an appropriate access method, depending on particular application needs, is a rather challenging problem. A spatial index library that can combine a wide range of indexing techniques under a common application programming interface can thus prove to be a valuable tool, since it will enable efficient application integration of a variety of structures in a consistent and straightforward way.

The major difficulty of such an undertaking is that most index structures have a wide range of distinctive characteristics, that are difficult to compromise under a common framework. For example, some structures employ data partitioning while others use space partitioning, some have rectangular node types while others have spherical node types, some are balanced while other are not, some are used only for indexing points while others are better for rectangles, lines, or polygons. Another important issue is that the index structures should provide functionality for exploiting the semantics of application-specific data types, through easy customization while making sure that meaningful queries can still be formulated for the specific data types. Moreover, it is crucial to adopt a common programming interface in order to promote reusability, easier maintenance and code familiarity, especially for large application projects where many developers are involved. The framework should capture the most important design characteristics, common to most structures, into a concise set of interfaces. This will help developers concentrate on other aspects of the client applications promoting, in this manner, faster and easier implementation. The interfaces should be easily extensible in order to address future needs without necessitating

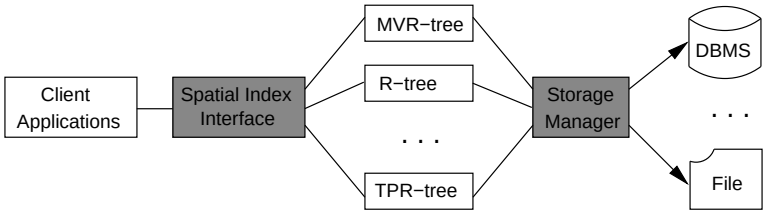

Figure 1: Architectural diagram of SaIL.

revisions to 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 at [1] and has been used successfully for both research and commercial purposes (e.g., [13, 19]).

The framework consists of four components:

1. The core library toolkit which provides basic functionality. For example, generic types like variants, implementation of utility classes like property sets, an exception class hierarchy tailored for spatial indices, etc.

2. The storage manager toolkit for supporting diverse storage requirements. This container is useful for decoupling the index structure implementation from the actual storage system, enabling proper encapsulation.

3. The spatial index interface which is a set of abstractions of the most common spatial index operations and related types. For example, interfaces for nodes, leafs, spatial data types, spatio-temporal queries, and more.

4. The concrete implementations of various spatial index structures. For example, variants of the R-tree, the MVR-tree, the TPR-tree, etc.

SaIL is designed to sit in-between the client application and the access methods that need to be interfaced by the client code. A simple architectural diagram is shown in Figure 1.

Next section presents some related work. Section 3 introduces the proposed framework. Section 4 discusses various popular index structures and how they can be easily integrated into the SaIL framework. Finally, Section 5 concludes the paper.

2 Related Work

The most relevant spatial index library to SaIL is the eXtensible and fleXible Library (XXL) [11]. XXL offers both low-level and high-level components for development and integration of spatial index structures like cursors, access to raw disk, a query optimizer, etc. Even though XXL is a superset of SaIL, it differs in three respects: First, SaIL offers a very concise, straightforward interface for querying arbitrary index structures in a uniform manner. In contrast, XXL querying interfaces are index specific (apart for join and aggregation queries). Second, SaIL offers more generic querying capabilities. Despite the fact that XXL can support a variety of advanced spatial queries (with the use of a framework for generalizing an incremental best-first search query strategy), nonconventional user defined queries have to be implemented by hand requiring modifications in all affected index structures. In contrast, SaIL offers an intuitive interface, utilizing well known design patterns, for formulating novel queries without having to revise the library in any way. Finally, SaIL provides the capability to customize query behavior during execution with the use of standardized design patterns. In contrast, in XXL this capability has to be supported explicitly by all index structure implementations.

GiST (for *Generalized Search Tree* [20]) is also relevant to this work. GiST is a framework that generalizes a height balanced, single rooted search tree with variable fanout. In essence, GiST is a *parameterized* tree that can be customized with user defined data types and user defined functions on these types which help guide the structural and searching behavior of the tree. Each node in the tree consists of a set of predicate/pointer pairs. Pointers are used to link the node to children nodes or data entries. Predicates are the user defined data types stored in the tree. The user, apart from choosing a predicate domain (e.g., the set of natural numbers, rectangles on a unit square universe, etc.), must also implement a number of methods (i.e., *consistent*, *union*, *penalty* and *pickSplit*) which are used internally by GiST to control the behaviour of the tree. By using a simple interface, GiST can support a wide variety of search trees and their corresponding

querying capabilities, including B-trees [9] and R-trees [18].

In order to support an even wider variety of structures, Aoki [2] proposed three additions to GiST. The GiST interface was augmented with multiple predicate support, which is useful for storing meta-data in the tree nodes. The tree traversal interface was also improved, so that the user can define complex search strategies. Finally, support for divergence control was added, so that a predicate contained in a parent node need not correspond to an accurate description of its subtree (for example an R-tree with relaxed parent MBRs that do not tightly enclose their children). Other extensions have also been proposed that modify the internal structure of GiST so that it can support very specialized indices, for instance, the TPR-tree [23] for moving objects.

Aref and Ilyas proposed the SP-GiST framework [3] which provides a novel set of external interfaces and original design, for furnishing a generalized index structure that can be customized to support a large class of spatial indices with diverse structural and behavioral characteristics. The generality of SP-GiST allows the realization of space and data driven partitioning, balanced and unbalanced structures. SP-GiST can support k-D-trees [5], tries [10, 15], quadtrees [14, 25] and their variants, among others.

Our work is orthogonal to XXL, GiST, SP-GiST and their variants. These libraries address the implementation issues behind new access methods by removing the burden of writing structural maintenance code from the developer. SaIL does not aim to simplify the development process of the index structures per se, but more importantly, the development of the applications that use them. *In that respect, it can be used in combination with all other index structure developer frameworks.* Employing SaIL is an easy process that requires the implementation of simple adapter classes that will make index structures developed with XXL, GiST, and SP-GiST compliant to the interfaces of SaIL, conflating these two conflicting design viewpoints. Ostensibly, existing libraries can be used for simplifying client code development as well — and share, indeed, some similarities with SaIL (especially in the interfaces for inserting and deleting elements from the indices) — but given that they are not targeted primarily for that purpose, they are not easily extensible (without the subsequent need for client code revisions), they do not promote transparent usage of diverse indices (since they use index specific interfaces), and they cannot be easily used in tandem with any other index library (existing or not). Finally, SaIL is the first library that introduces functionality for supporting temporal and spatio-temporal indices, through specialized

Table 1: Summary of novel features of SaIL ('$\surd$' full, '$\times$' partial, '-' none).

Features	SaIL	XXL	GiST	SP-GiST
Query execution customization	$\surd$	$\times$	$\times$	$\times$
Query strategy customization	$\surd$	$\times$	-	-
Generic shape abstractions	$\surd$	-	-	-
Storage management decoupling	$\surd$	$\times$	$\times$	$\times$
Index operation customization	$\surd$	-	-	-
Temporal/Spatio-temporal interfaces	$\surd$	-	-	-

capabilities that all other libraries are lacking. A table summarizing the novel features of SaIL (that will be explained in more detail subsequently) appears in Table 1.

3 Spatial Index Library Architecture

In this section we present the most important concepts behind SaIL's design decisions using various examples. Figure 2 summarizes the UML notation used in the text and diagrams. When referring to a specific design pattern we use the definitions of Gamma et al. [17]. For convenience, some design pattern descriptions (quoted from [17]) are listed in Table 2.

Table 2: Design Pattern Descriptions

Name	Description
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.

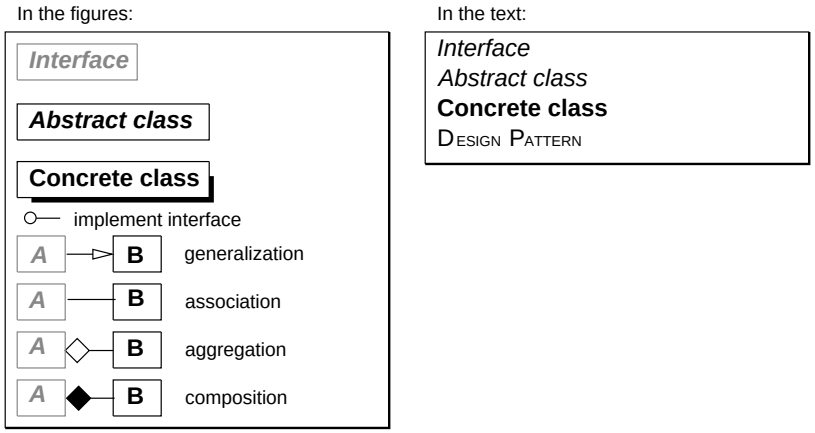

Figure 2: Notation used in the paper.

3.1 The Core Toolkit

The abstract and concrete classes offered by the core toolkit are shown in Figure 3 (note that this and the remaining figures adopt the diagrammatic notation shown in Figure 2). 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. For example, adding and handling the initialization of new properties to an object can be achieved without modifying its constructor, if a **PropertySet** is used as one of the constructor's arguments.

An index specific *Exception* class hierarchy is provided which helps promote the use of exception handling in client code, in a structured manner. Some basic interfaces representing essential object properties are also defined. For example interfaces for serializing and comparing objects, defining streams/iterators on object collections, etc. Other helper classes are provided as well, representing open/closed intervals, random number generators, resource usage utilities, etc. (some utilities are left out of this figure).

An important utility class provided by the core toolkit is **ExternalSort**. External sorting is needed for sorting very large relations that cannot be stored in main memory, an operation that is

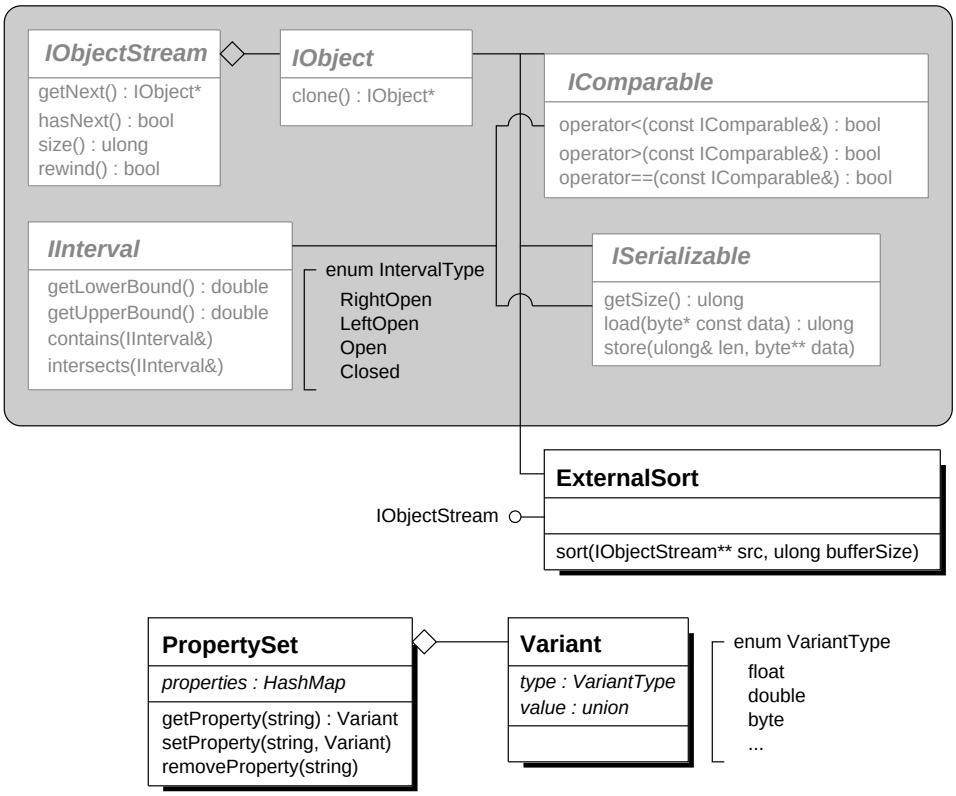

Figure 3: The Core Toolkit.

essential for bulk loading an index structure. By using the *IObject*, *IComparable* and *ISerializable* interfaces, **ExternalSort** is a generic sorting utility that can sort entries of any user defined data type in a very robust and straightforward manner. The caller needs to define an object stream that provides the sorter with objects of type *IObject* (using the *IObjectStream* interface), in random sequence. Then, **ExternalSort** iterates over the input objects is sorted order, as implied by a user provided comparator implementation. Sorting is accomplished by using temporary files on secondary storage. Conveniently, **ExternalSort** implements the *IObjectStream* interface itself and, thus, can be used as a PROXY in place of an *IObjectStream*, with the only difference that the stream appears to be in sorted order.

3.2 The Storage Manager Toolkit

A critical part of spatial indexing tools is the storage manager, which should be versatile, very efficient and provide loose coupling. Clients that must persist entities to secondary storage should be unaware of the underlying mechanisms, in order to achieve proper encapsulation. Persistence

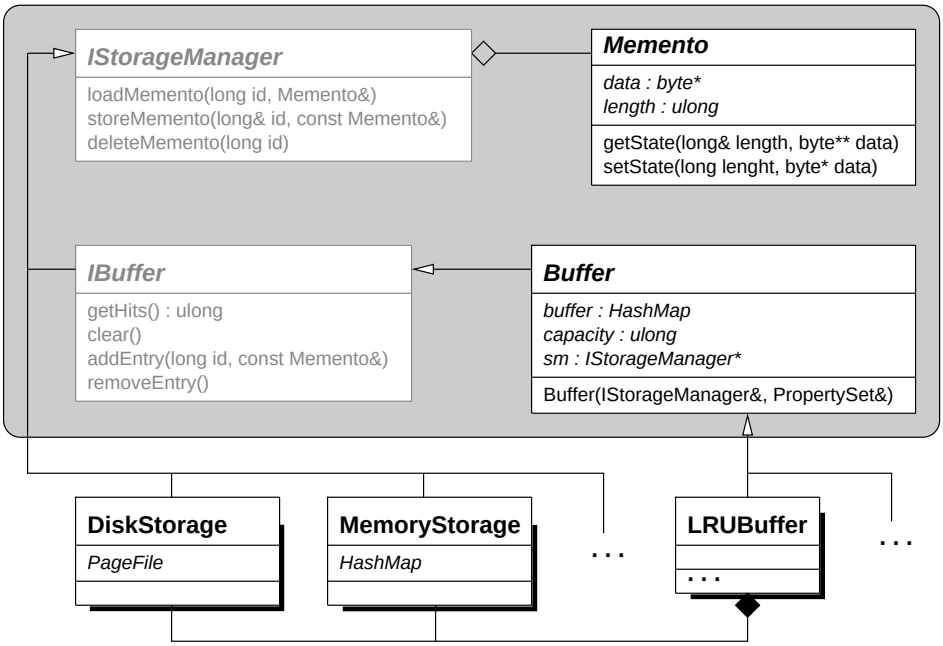

Figure 4: The Storage Manager Toolkit.

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, without the need to update the client code.

The storage manager toolkit is shown in Figure 4. 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 return an identifier that can be used to retrieve the object's state at a later time. Two concrete implementations of the *IStorageManager* interface are already provided. A main memory manager that uses a hash table and a disk based page file.

This architecture allows multiple layers of storage management facilities to be streamlined one after the other. A simple example is the need to store data over a network. A simple adapter class can be implemented that sends the data over the network with the proper format expected by the remote computer that uses a second adapter class to persist the data. Another example is the implementation of a relational based index structure. An adapter can be embedded on an existing implementation, converting the data to be persisted into appropriate SQL statements

that store them in a relational table (e.g., as BLOBs).

The *IBuffer* interface provides basic buffering functionality. It declares the two most important operations of a buffer: adding and removing an entry. The *Buffer* abstract class provides a default implementation for the major operations of a buffer component. For convenience, a concrete implementation of a Least Recently Used buffering policy is already provided. Using the *IBuffer* interface is straightforward; it acts as a proxy between an actual storage manager and a client, buffering entries as it sees fit. The client instantiates a concrete buffer class, provides the buffer with a reference to the actual storage manager and finally associates the index structure (or any other component that will use the storage manager) with a reference to the buffer. Conveniently, the callers are unaware that buffering is taking place by assuming that a direct interface with the actual storage manager is used. This architecture provides sufficient flexibility to alter buffering policies at runtime, add new policies transparently, etc.

3.3 The Spatial Index Interface

Spatial access methods are used for indexing 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 5) 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 [18] can be accomplished by calling the *IShape.getMBR* method to obtain the minimum bounding region of the polygon. As long as the user defined polygon class returns a proper MBR representation, the R-tree remains unaware of the actual details of the polygon class, internally — all it needs to be aware of is the *IShape.getMBR* contract. Complex shapes and combinations of shapes can be represented by composing a number of *IShapes* under one class and, hence, can be handled in a uniform manner as all other classes of type *IShape*. Various methods useful for comparing *IShapes* (like *contains*, *intersects*, etc.) are also specified. In addition, the *IShape* interface can handle shapes with arbitrary dimensionality, so that multi-dimensional indices can be supported.

The *ITimeShape* interface extends *IShape* with methods for comparing shapes with temporal predicates. An *ITimeShape* has temporal extents, acquired by implementing the *IInterval* inter-

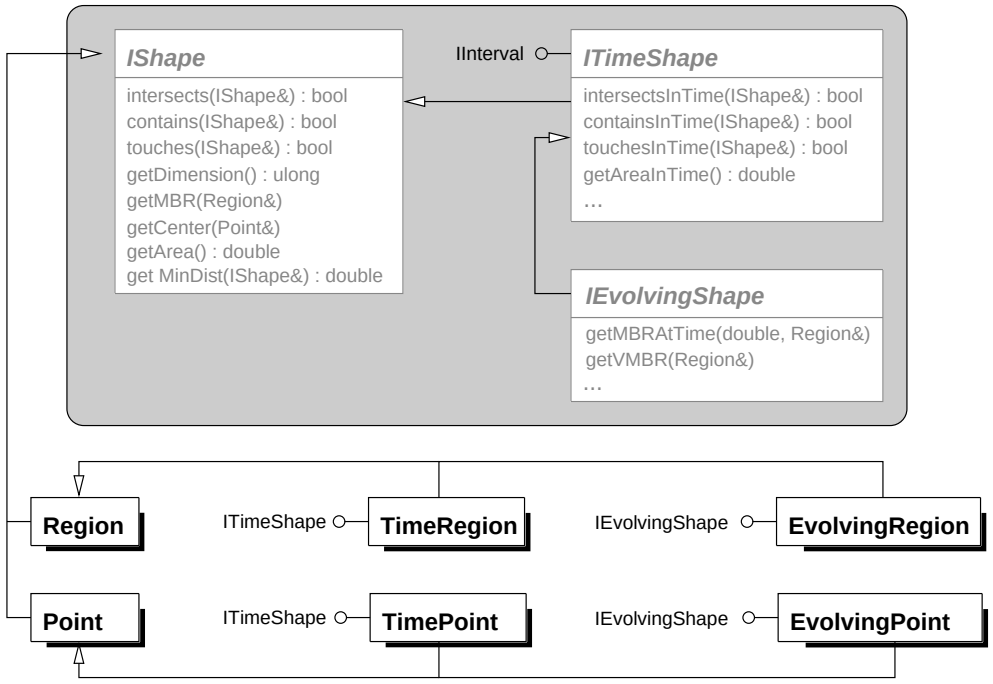

Figure 5: The shape interfaces.

face. This abstraction is useful for handling spatio-temporal indices, where the indexed objects are associated with insertion and deletion times, as well as for specifying spatial queries with temporal restrictions.

Furthermore, to cover special cases where shapes evolve over time, the *IEvolvingShape* interface is provided. An evolving shape can be represented by associating a velocity vector to every vertex of the representation and, in addition, it can be approximated by an evolving MBR. Special methods must also be implemented for computing intersections, area, volume, and other characteristics over time. The *IEvolvingShape* interface is derived from *ITimeShape* and, thus, such shapes can be associated with temporal extents during which the evolution is taking place.

Concrete implementations of basic shapes are provided for convenience. These shapes provide basic functionality, like computing intersections, areas, temporal relations, etc.

An essential capability of a generic index manipulation 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 from client code. For example, querying functions should return iterators (i.e., enumerations or cursors) over well-defined data elements, irrespective of what kind of structures they operate on. In general, a hierarchical index structure is composed of a number

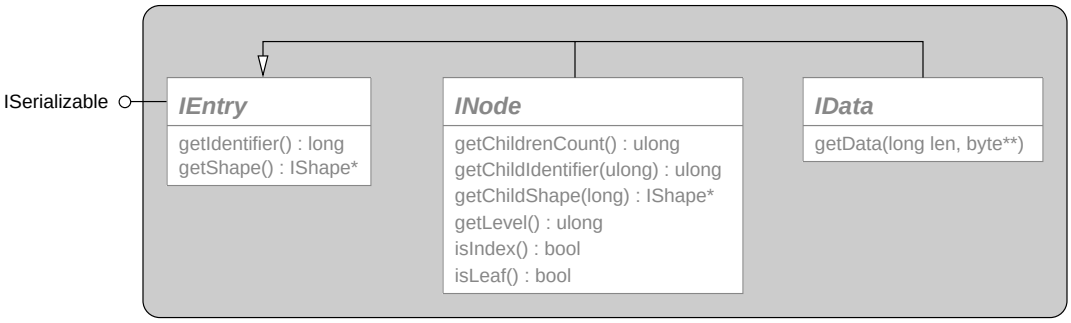

Figure 6: The index element interface.

of nodes and a number of data entries. Every node has a specific shape, identifier and level. Nodes are further divided into index nodes and leaves. Data entries are either simple pointers to the actual data representations on disk (a secondary index), or the actual data themselves (a primary index). The data elements can be either exact shape representations or more generic meta-data associated with a specific index entry. These concepts are represented by using the following hierarchy: *IEntry* is the most basic interface for a spatial index entry; its basic members are an identifier and a shape. *INode* (that inherits from *IEntry*) represents a generic tree node; its basic members are the number of children, the tree level, and a property specifying if it is an index node or a leaf. The *IData* interface represents a data element and 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. All methods take *IShapes* as arguments and every shape is associated with a user defined identifier that enables convenient referencing of objects. Below we describe each method in more detail.

The *insertData* method is used for inserting new entries into an index. It 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 (useful for implementing primary indices). The *deleteData* method locates and deletes an object already contained in an index. It accepts the *IShape* to be deleted and its object identifier. (The *IShape* argument is necessary since spatial indices cluster objects according to their spatial characteristics and not their identifiers.)

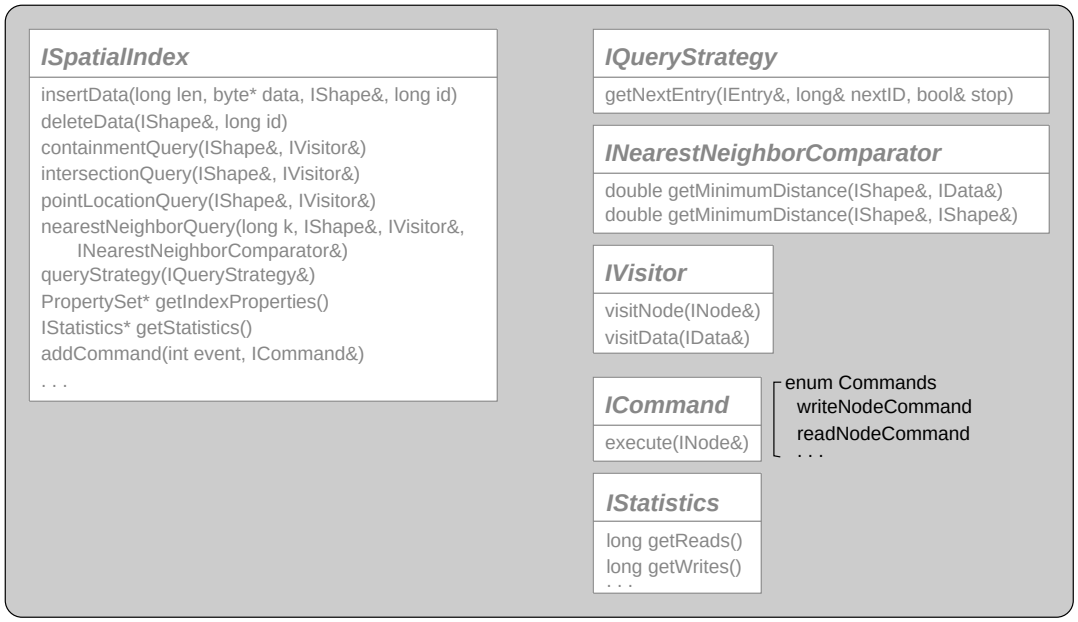

Figure 7: The generic spatial index interface.

The query methods take the 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 range search algorithm decides if a node should be examined by calling the query *intersects* predicate on a candidate node 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, for the *IShape* objects used as arguments to the querying methods.

In order to provide advanced customization capabilities 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 index and leaf nodes 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 (an interactive k-nearest neighbor query is a good example, where the user can stop the

execution of the query after an adequate number of answers has been discovered). Another useful example of the VISITOR pattern is tallying the number of query I/Os. By counting the number of times each visit method has been called, it is possible to count the number of index nodes, leaf nodes, or "any" level nodes that were accessed for answering a query (the *visitNode* method returns an *INode* reference when called, which provides all necessary information on the specific node accessed, e.g., its level, shape, etc.). A tallying visitor is presented in Algorithm 1. A different example is visualizing the progress of the query. As the querying algorithm proceeds, the visitor can draw the last accessed node or data element on the screen. An access method can support the *IVisitor* interface simply by guaranteeing that all query algorithms call the *visitNode* and *visitData* methods of *IVisitor*, every time a node or a data entry is accessed while searching the structure. Thus, supporting the *IVisitor* interface requires a very simple, inexpensive procedure.

Algorithm 1 *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();
  }
};
```

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.

To illustrate the simplicity of supporting the *IVisitor* interface from the querying methods of a spatial index implementation, the actual implementation of a range query algorithm of a

hierarchical structure that supports all of the aforementioned features is shown in Algorithm 2.

Algorithm 2 Range query method implementation that supports the *IVisitor* interface.

```
void rangeQuery(const IShape& query, IVisitor& v) {
  stack<NodePtr> st;
  Node* root = readNode(m_rootID);

  if (root->m_children > 0 && query.intersects(root->m_nodeBoundary)) st.push(root);

  while (! st.empty()) {
    Node* n = st.top(); st.pop();

    if (n->isLeaf()) {
      v.visitNode(*n);

      for (unsigned long cChild = 0; cChild < n->m_children; cChild++) {
        if (query.intersects(n->m_childBoundary[cChild])) {
          v.visitData(n->m_childData[cChild]);
        }
      }
    } else {
      v.visitNode(*n);

      for (unsigned long cChild = 0; cChild < n->m_children; cChild++)
        if (query.intersects(n->m_childBoundary[cChild]))
          st.push(readNode(n->m_childIdentifier[cChild]));
    }
  }
}
```

An even larger degree of customization is provided for the nearest neighbor query method. Since different applications use diverse distance measures to identify nearest neighbors (like the Euclidean distance, and others), the *nearestNeighborQuery* method 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 application specific changes to the library. In reality, a nearest neighbor comparator is essential, since in order to find the actual nearest neighbors of a query, the query has to be compared with each candidate's exact representation so that an exact distance can be computed. Since most spatial index structures store object approximations in place of the real objects (e.g., R-trees store MBRs), internally they make decisions based on approximate distance computations and, hence, cannot identify the exact nearest neighbors. One way to overcome this weakness, is to

let the index load the actual objects from storage and compute the real distances only when appropriate. Albeit, this would break encapsulation since loading the actual objects implies that the index has knowledge about the object representations. Alternatively, the user can provide a nearest neighbor comparator that implements a method for comparing index approximations (e.g., MBRs) with the actual objects (e.g., polygons). Method *getMinimumDistance* is used for that purpose.

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 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 Algorithm 3 (the example requires less than 15 lines of code). Another example for computing the total space indexed by an R-tree is shown in Algorithm 4. Many other possible uses of the query strategy pattern exist.

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 parameterized 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,

Algorithm 3 Breadth-first traversal of index nodes.

```
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;
    }
  }
};
```

Algorithm 4 Finding the total indexed space.

```
class MyQueryStrategy : public IQueryStrategy
{
public:
  Region m_indexedSpace;

public:
  void getNextEntry(IEntry& e, long& nextID, bool& stop) {
    // stop after the root.
    stop = true;

    IShape* ps;
    entry.getShape(&ps);
    ps->getMBR(m_indexedSpace);
    delete ps;
  }
};
```

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. Another example is the need to track specific index entries and be able to raise alerts whenever they take part in splitting or merging operations, or they get relocated a new disk page. 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.

4 Test-case Scenarios

The task of implementing robust index structures raises a number of important design issues. We argue that trying to make an index structure implementation compatible with the SaIL framework helps eliminate many design issues which have been resolved in advance. In addition, SaIL's interfaces help incorporate index structures very easily into existing applications, especially after developers become familiar with its basic features and design pattern approaches. To test our claim we implemented a classic spatial index, namely the R-tree [18], and two popular spatio-temporal structures, the MVR-tree [22, 21, 27] and the TPR-tree [23]. With this choice of test-cases we show the versatility and applicability of the framework to both traditional and non-traditional structures. The MVR-tree is a multi-version structure proposed for archiving and querying "past" versions of an R-tree, as it evolves through updates. In contrast, the TPR-tree has been proposed for querying "future" positions of moving objects, based on their current positions and velocity vectors. Below we discuss various implementation issues of the three test-cases.

4.1 R-tree

The R-tree design process is simplified substantially in a very intuitive manner. The *IEntry* hierarchy directs the decomposition of tree entries into *Node* and *Data* classes, implementing *INode* and *IData* respectively. In turn, *Node* can be extended into specialized **Index** and **Leaf** classes. By design, all tree entries point to other entries of type *IEntry*. Thus, since both index and leaves operate on children of the same type, common functionality (like node splitting algorithms,

deletion algorithms, update propagation, etc.) shared by both, are implemented in the generic class *Node*. Defining **Index** and **Leaf** allows variable node fanout among these fundamentally different tree levels as well as specializing leaves with useful methods that act on entries of type *IData*.

Persisting a node (index or leaf) requires a simple conversion of its state to a byte array and the instantiation of an appropriate *Memento* class. Storage management and buffering decisions are differed to the client code, which can use the storage management facilities of the framework (or any other framework, like SHORE [8]) directly. The tree internally only needs to use the *ISerializable* and *IStorageManager* interfaces, simplifying substantially node storage issues.

Customizing various internal operations of the R-tree (like writing, reading, deleting, splitting nodes, inserting and deleting data, etc.), is achieved by using several *ICommand* lists and the *addCommand* method of *ISpatialIndex*, thus obscure call-back functions are avoided and multiple different commands can be executed per operation.

R-tree construction is simplified substantially with the use of **PropertySets**. Only one constructor and a list of property strings should be exported by the index. It is up to the user to initialize an appropriate property set or use the default values. The list of properties can grow easily according to future needs.

An important part of the framework is committing to the *ISpatialIndex* facade methods. The update methods accept *IShapes* and convert them internally to minimum bounding boxes (**Regions**) by using the $getMBR$ method of *IShape*. The query methods use the query shape $intersect$ and $contain$ predicates to guide the search of the tree without any knowledge of the actual shapes involved (thus any type of shapes can be used in an intuitive manner as long as they implement these predicates). Also, the query methods respect the *IVisitor* interface simply by calling the $visitNode$ and $visitData$ methods on all accessed entries. By conforming to these two simple interfaces the index can support a wide range of user customizable queries as well as achieving proper encapsulation and avoid contaminating the implementation with application specific logic.

4.2 MVR-tree

The second test-case illustrates how SaIL can support multi-version access methods. A data structure is called multi-version (or "persistent" [12, 24, 4]) if it maintains all its past versions. Typical data structures are single-version (or "ephemeral") in the sense that an update (insertion or deletion) creates a new version of the data structure while its previous version is discarded.

The MVR-tree is a multi-version structure that efficiently stores the evolution of an R-tree over a sequence of updates. Consider for example an ephemeral R-tree (version V_0) indexing a collection of objects. Assume that a sequence of updates is applied on this tree. For simplicity, assume that each operation results in a new version of the R-tree. That is, updates $u_1, u_2, ..., u_n$ create versions $V_1, V_2, ..., V_n$. Note that a new version is created by updating the previous version (i.e., a *linear versioning* is assumed). The aim is to maintain all individual versions that the R-tree ever obtained. Querying past states of a structure provides additional functionality since it allows access to the objects indexed by the structure at a particular version.

One simplistic, but rather inefficient approach to maintain all versions of an evolving data structure is to store a complete snapshot of the structure at every update. This, however, would result in prohibitively large (quadratic) space utilization. Instead, the MVR-tree embeds all past versions of the evolving R-tree into its structure with little space overhead (linear to the number of updates in the R-tree evolution).

Introduction to the MVR-tree. The MVR-tree is a directed acyclic graph with multiple root nodes each of which is responsible for recording a consecutive part of the ephemeral R-tree evolution. Thus a root in the MVR-tree is associated with a disjoint version interval of the R-tree evolution. A query about a particular version V_q would follow only the root that contains V_q.

Data records in the leaf nodes maintain the evolution of the ephemeral R-tree data objects. Since past states need to be recorded, object records are extended to include "lifetime" version intervals (V_i, V_j), where V_i is the version when this object was inserted in the R-tree and V_j the version when it was deleted (if ever) from the R-tree ($V_j > V_i$). A physical insertion of an object in version V_i of the R-tree corresponds to inserting a record for this object in the MVR-tree, accompanied by interval $(V_i, *)$ (here $*$ represents the yet unknown end-version for this object). If this object is physically deleted from the R-tree at a later version V_j, the $*$ in its version interval

is replaced by V_j (i.e., it corresponds to a "logical" deletion in the MVR-tree). Similarly, index records in the directory nodes maintain the evolution of the corresponding index records of the ephemeral R-tree and are also augmented with version intervals.

A data record is termed "alive" for all versions included in its version interval. With the exception of root nodes, a leaf (or index) node is "alive" for all versions that it has at least $B \cdot P_v$ alive records, where $0 < P_v < 0.5$ is a tree parameter and B is the node capacity. This implies that at least half of the entries contained in any node that is alive, should also be alive. This requirement enables clustering objects that are alive at a given version in a small number of nodes, which in turn will minimize the query I/O. This is achieved by using novel merging and splitting policies during updates.

Consider an update at version V. The MVR-tree is searched to locate the target leaf node where the update is to be applied. This search is carried out using the version intervals of the index and leaf nodes (they should contain V), while traversing the structure. *Non-structural* are those updates which are handled within the target leaf node. Otherwise, if an update creates a new node, it is termed *structural* [4]. A structural update is triggered by (1) an insertion if the target leaf node already has B records (*node overflow*), or, by (2) a deletion that lets the target node with less than $B \cdot P_v$ alive records (*weak version underflow*).

Node overflow and weak version underflow require special handling. Overflows cause a *split* on the target leaf node. Splitting a node A at version V is performed by creating a new node A' and copying all the alive records from A to A'. Node A is considered *dead* after version V. To avoid having frequent structural changes on node A', the number of alive entries that it contains must be in the range $[B \cdot P_{svu}, B \cdot P_{svo}]$, where P_{svu} and P_{svo} are predetermined constants. This allows a constant number of non-structural changes on A' before a new structural change occurs. If A' has more than $B \cdot P_{svo}$ alive records a *strong version overflow* occurs; the node will have to be *key-split* into two new nodes. A key-split does not take object version lifetimes into account. Instead, it splits the entries according to their spatial characteristics (e.g., by using the R*-Tree splitting algorithm which tries to minimize spatial overlap). On the other hand, if A' has less than $B \cdot P_{svu}$ alive records a *strong version underflow* occurs; the node will have to be merged with a sibling node before it can be incorporated into the structure ([22, 27] discuss various merging policies in detail).

Using SaIL to implement the MVR-tree. Since, in essence, the MVR-tree is a structure that encapsulates a combination of many R-trees it can be implemented using the SaIL framework as a direct extension of the R-tree. Few modifications however are necessary, relating to handling insertions, deletions and queries with "version-aware" shapes, as well as to the new merging and splitting policies.

Since most of these operations act on nodes, the R-tree *Node* class can be extended, and a few methods can be overridden, to support the modified splitting and the new merging policies. Inserting data into the tree requires specifying the associated version lifetime of the data. Modifying the *ISpatialIndex* facade is obviously not an option. Instead, we define extensions of all concrete shapes, by implementing the *IInterval* interface. Every shape inserted in the MVR-tree, as a constraint, should implement this interface — a commitment that can be verified at runtime. The querying methods do not have to be modified either. A query shape should also include an associated query version interval. For these purposes the *ITimeShape* interface has been defined, which extends *IShape* to implement *IInterval*. Intuitively, **Points** and **Regions** are extended into **VersionPoints** and **VersionRegions** respectively (which are specializations of *ITimePoint* and *ITimeRegion* that can handle open ended intervals, needed for the MVR-tree). This modification fits nicely into the SaIL framework; the new interfaces and classes are simple extensions of already existing classes, thus, interface definitions need not be modified (something that would create incompatibilities with existing clients). Code reuse is promoted by extending existing shapes into their "version-aware" counterparts.

An interesting problem is how to traverse the complete MVR-tree using the SaIL framework, since the graph has multiple roots. A simple modification would extend the *ISpatialIndex* facade with methods that can initiate searches on specific roots, a solution that is not preferable since other access methods would need to be modified to support the new facade. Moreover, the *queryStrategy* method initiates a search by returning the "only" root of a tree at the first iteration. An elegant and very simple solution in case of multiple roots is to augment the tree structure with a sentinel node that points to all the roots (i.e., a list of roots). Hence, the tree can still take advantage of the *queryStrategy* pattern as long as users that need this functionality are aware that the sentinel node is returned as the first iteration of the method (which depends on proper documentation).

4.3 TPR-tree

The TPR-tree is a spatio-temporal index structure that indexes moving objects, namely d-dimensional rectangles that follow linear movements in time. The structure is a direct extension of R-trees, with the only difference being that objects are grouped into time-parameterized MBRs — each vertex of a time-parameterized MBR is associated with a velocity vector that helps compute the position of the MBR in the future. As time progresses the MBR evolves by shrinking or enlarging independently in all dimensions. The idea is that a time-parameterized MBR bounds the evolving objects that it contains at all times in the future, by taking into account the movements of its children, inside some pre-specified horizon. Objects are grouped into nodes according to both their spatial location and velocity characteristics.

The TPR-tree can be directly derived from the R-tree, since structurally both indices are the same. Two important changes have to be made: (1) The *Node* class has to be extended to support splitting policies for time-parameterized MBRs that take velocities into account; (2) In addition, special *IShapes* with geometric properties of evolving shapes need to be defined. For that purpose the *IEvolvingShape* interface is provided as a direct extension of the *ITimeShape* interface. **EvolvingPoint** and **EvolvingRegion** act as the time-parameterized objects of the structure. Any shape that implements *IEvolvingShape* can be inserted into the structure as long as the $getVMBR$ method is supported. The tree internally approximates the object into an evolving MBR by verifying that the object implements *IEvolvingShape* at runtime.

All classes needed to support the TPR-tree are extensions of already defined interfaces. Notice the intuitive and progressive evolution of the *IShape* hierarchy from simple shapes indexed in a spatial index, to time-aware shapes indexed into a spatio-temporal index, to evolving time-aware shapes indexed into a structure for moving objects. SaIL's interface hierarchies are apt to progressive and straightforward extensions, according to application needs. In addition, methods defined for existing objects can be used directly when applicable, in new structures. For instance, the **EvolvingRegion** class is derived from **TimeRegion**, which means that all time-aware methods do not have to be re-implemented. Other methods that need to take into account velocity vectors can be added and implemented directly into the new class, without affecting other code.

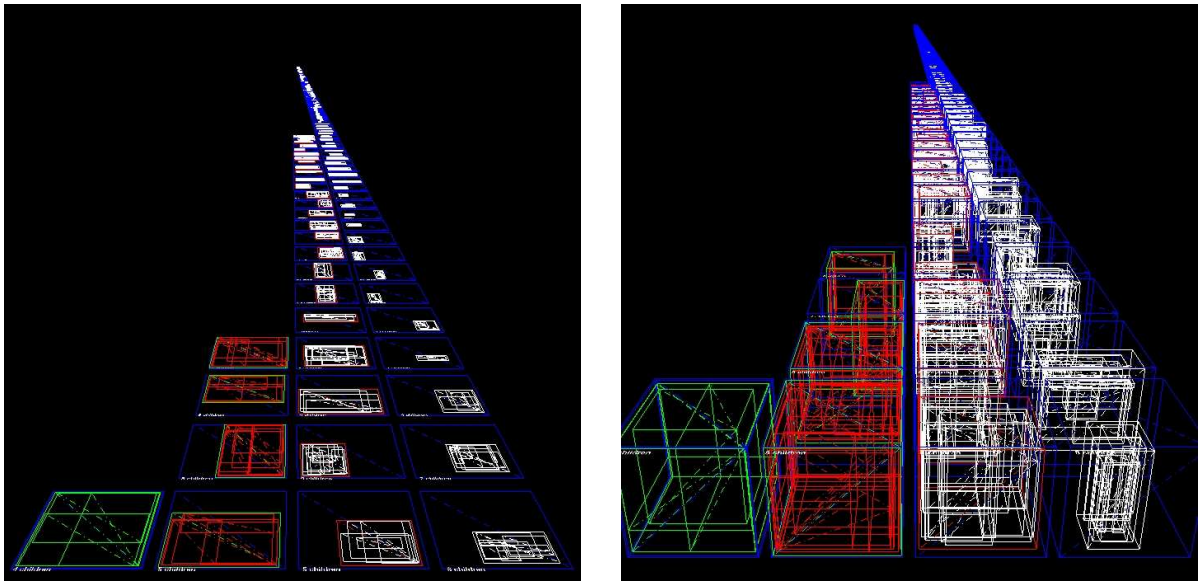

Figure 8: Visualizing 2- and 3-dimensional R-trees.

4.4 R-tree Visualization

Finally, as a proof of concept, we present a simple application for visualizing R-trees. We have implemented a Java applet that can read any 2- or 3-dimensional R-tree created with our library, and draw its underlying structure on screen (Figure 8).

The applet uses the Java3D library to draw on the screen, and the basic facilities provided by SaIL to access and traverse the index structure. We implemented a simple query strategy that traverses the tree and returns all the essential information about tree nodes, sequentially. Interfacing the tree is extremely simple and requires only few lines of code. The demo can be downloaded from [1].

5 Conclusions

We presented an extensible spatial index framework for developers of spatial and spatio-temporal applications. We argue that the proposed framework will help developers incorporate indexing techniques with greater ease into existing applications. The framework is based on well documented design patterns that promote reusability and improved code quality. SaIL has already been used in both research (UCR) and industry (ESRI) environments and we believe it will be very helpful for the GIS community. A sample implementation in C++ and Java can be downloaded

freely from [1].

References

[1] SaIL. http://spatialindexlib.sourceforge.net.

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

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

[4] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-Tree. *The VLDB Journal*, 5(4):264–275, 1996.

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

[6] S. Berchtold, D. A. Keim, and H. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. of Very Large Data Bases (VLDB)*, pages 28–39, 1996.

[7] C. Böm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[8] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *Proc. of ACM Management of Data (SIGMOD)*, pages 383–394, 1994.

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

[10] R. de la Briandais. File searching using variable length keys. In *Proc. of the Western Joint Computer Conference*, pages 295–298, 1959.

[11] 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 (VLDB)*, pages 39–48, 2001.

[12] J. Driscoll, N. Sarnak, D. Sleator, and R.E. Tarjan. Making data structures persistent. In *Proc. of ACM Symposium on Theory of Computing (STOC)*, 1986.

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

[14] R. A. Finkel and J. L. Bentley. Quad Trees, a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[15] E. Fredkin. Trie memory. *Communications of the ACM (CACM)*, 3(9):490–499, 1960.

[16] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[17] 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.

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

[19] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Efficient indexing of spatiotemporal objects. In *Proc. of Extending Database Technology (EDBT)*, pages 251–268, 2002.

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

[21] G. Kollios, V.J. Tsotras, D. Gunopulos, A. Delis, and M. Hadjieleftheriou. Indexing animated objects using spatiotemporal access methods. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 13(5):758–777, 2001.

[22] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing access methods for bitemporal databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 10(1):1–20, 1998.

[23] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. *SIGMOD Record*, 29(2):331–342, 2000.

[24] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *Communications of the ACM (CACM)*, 31(2):158–221, 1999.

[25] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

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

[27] Y. Tao and D. Papadias. MV3R-Tree: A spatio-temporal access method for timestamp and interval queries. In *Proc. of Very Large Data Bases (VLDB)*, pages 431–440, 2001.