libspatialindex Documentation

Release 1.9.0

Marios Hadjieleftheriou

Feb 04, 2019

CONTENTS

CHAPTER

ONE

INSTALLATION

libspatialindex supports building and installation using a couple of different methods. The primary method on unix-like systems is the Autotools-style build. Alternatively, a CMake-derived build system is also available for building libspatialindex as of the 1.7.0 release

1.1 Autotools

First run *autogen.sh* to generate the configure scripts. By default include files and library files will be installed in */usr/local*. If you would like to use a different installation directory (e.g., in case that you do not have root access) run the configure script with the –prefix option:

```
[hobu@fire libspatialindex]$ ./configure --prefix=/home/marioh/usr
```

Make the library:

```
[hobu@fire libspatialindex]$ make
```

Install the library:

```
[hobu@fire libspatialindex]$ make install
```

1.2 CMake

The CMake scenario for building libspatialindex is also quite simple.

```
[hobu@fire libspatialindex]$ cmake -DCMAKE_INSTALL_PREFIX=/home/marioh/usr .
```

```
[hobu@fire libspatialindex]$ make
```

```
[hobu@fire libspatialindex]$ make install
```

CHAPTER

TWO

LIBRARY OVERVIEW

The library currently consists of six packages:

1. The core spatialindex utilities.
2. The storagemanager files.
3. The spatialindex interfaces.
4. The rtree index.
5. The mvrtree index.
6. The tprtree index.

I will briefly present the basic features supported by each package. For more details you will have to refer to the code, for now.

2.1 Spatial Index Utilities

To provide common constructors and uniform initialization for all objects provided by the library a PropertySet class is provided. A PropertySet associates strings with Variants. Each property corresponds to one string.

A basic implementation of a Variant is also provided that supports a number of data types. The supported data types can be found in SpatialIndex.h

PropertySet supports three functions:

1. getProperty returns the Variant associated with the given string.
2. setProperty associates the given Variant with the given string.
3. removeProperty removes the specified property from the PropertySet.

A number of exceptions are also defined here. All exceptions extend Exception and thus provide the what() method that returns a string representation of the exception with useful comments. It is advisable to use enclosing try/catch blocks when using any library objects. Many constructors throw exceptions when invalid initialization properties are specified.

A general IShape interface is defined. All shape classes should extend IShape. Basic Region and Point classes are already provided. Please check Region.h and Point.h for further details.

2.2 Storage Manager

The library provides a common interface for storage management of all indices. It consists of the IStorageManager interface, which provides functions for storing and retrieving entities. An entity is viewed as a simple byte array;

hence it can be an index entry, a data entry or anything else that the user wants to store. The storage manager interface is generic and does not apply only to spatial indices.

Classes that implement the IStorageManager interface decide on how to store entities. simple main memory implementation is provided, for example, that stores the entities using a vector, associating every entity with a unique ID (the entry's index in the vector). A disk based storage manager could choose to store the entities in a simple random access file, or a database storage manager could store them in a relational table, etc. as long as unique IDs are associated with every entity. Also, storage managers should implement their own paging, compaction and deletion policies transparently from the callers (be it an index or a user).

The storeByteArray method gets a byte array and its length and an entity ID. If the caller specifies NewPage as the input ID, the storage manager allocates a new ID, stores the entity and returns the ID associated with the entity. If, instead, the user specifies an already existing ID the storage manager overwrites the old data. An exception is thrown if the caller requests an invalid ID to be overwritten.

The loadByteArray method gets an entity ID and returns the associated byte array along with its length. If an invalid ID is requested, an exception is thrown.

The deleteByteArray method removes the requested entity from storage.

The storage managers should have no information about the types of entities that are stored. There are three main reasons for this decision:

1. Any number of spatial indices can be stored in a single storage manager (i.e. the same relational table, or binary file, or hash table, etc., can be used to store many indices) using an arbitrary number of pages and a unique index ID per index (this will be discussed shortly).
2. Both clustered and non-clustered indices can be supported. A clustered index stores the data associated with the entries that it contains along with the spatial information that it indexes. A non-clustered index stores only the spatial information of its entries. Any associated data are stored separately and are associated with the index entries by a unique ID. To support both types of indices, the storage manager interface should be quite generic, allowing the index to decide how to store its data. Otherwise clustered and non-clustered indices would have to be implemented separately.
3. Decision flexibility. For example, the users can choose a clustered index that will take care of storing everything. They can choose a main memory non-clustered index and store the actual data in MySQL. They can choose a disk based non-clustered index and store the data manually in a separate binary file or even in the same storage manager but doing a low level customized data processing.

Two storage managers are provided in the current implementation:

1) MemoryStorageManager

2) DiskStorageManager

2.2.1 MemoryStorageManager

As it is implied be the name, this is a main memory implementation. Everything is stored in main memory using a simple vector. No properties are needed to initialize a MemoryStorageManager object. When a MemoryStorageManager instance goes out of scope, all data that it contains are lost.

2.2.2 DiskStorageManager

The disk storage manager uses two random access files for storing information. One with extension .idx and the other with extension .dat.

A list of all the supported properties that can be provided during initialization, follows:

Property	Type	Description
FileName	VT_PCHAR	The base name of the file to open (no extension)
Overwrite	VT_BOOL	If Overwrite is true and a storage manager with the specified filename already exists, it will be truncated and overwritten. All data will be lost.
PageSize	VT_ULONG	The page size to use. If the specified filename already exists and Overwrite is false, PageSize is ignored.

For entities that are larger than the page size, multiple pages are used. Although, the empty space on the last page is lost. Also, there is no effort whatsoever to use as many sequential pages as possible. A future version might support sequential I/O. Thus, real clustered indices cannot be supported yet.

The purpose of the .idx file is to store vital information like the page size, the next available page, a list of empty pages and the sequence of pages associated with every entity ID.

This class also provides a flush method that practically overwrites the .idx file and syncs both file pointers.

The .idx file is loaded into main memory during initialization and is written to disk only after flushing the storage manager or during object destruction. In case of an unexpected failure changes to the storage manager will be lost due to a stale .idx file. Avoiding such disasters is future work.

2.3 SpatialIndex Interfaces

A spatial index is any index structure that accesses spatial information efficiently. It could range from a simple grid file to a complicated tree structure. A spatial index indexes entries of type IEntry, which can be index nodes, leaf nodes, data etc. depending on the structure characteristics. The appropriate interfaces with useful accessor methods should be provided for all types of entries.

A spatial index should implement the ISpatialIndex interface.

The containmentQuery method requires a query shape and a reference to a valid IVisitor instance (described shortly). The intersectionQuery method is the same. Both accept an IShape as the query. If the query shape is a simple Region, than a classic range query is performed. The user though has the ability to create her own shapes, thus defining her own intersection and containment methods making possible to run any kind of range query without having to modify the index. An example of a trapezoidal query is given in the regressiontest directory. Have in mind that it is the users responsibility to implement the correct intersection and containment methods between their shape and the type of shapes that are stored by the specific index that they are planning to use. For example, if an rtree index will be used, a trapezoid should define intersection and containment between itself and Regions, since all rtree nodes are of type Region. Hence, the user should have some knowledge about the index internal representation, to run more sophisticated queries.

A point location query is performed using the pointLocationQuery method. It takes the query point and a visitor as arguments.

Nearest neighbor queries can be performed with the nearestNeighborQuery method. Its first argument is the number k of nearest neighbors requested. This method also requires the query shape and a visitor object. The default implementation uses the getMinimumDistance function of IShape for calculating the distance of the query from the rectangular node and data entries stored in the tree. A more sophisticated distance measure can be used by implementing the INearestNeighborComparator interface and passing it as the last argument of nearestNeighborQuery. For example, a comparator is necessary when the query needs to be checked against the actual data stored in the tree, instead of the rectangular data entry approximations stored in the leaves.

For customizing queries the IVisitor interface (based on the Visitor pattern [gamma94]) provides callback functions for visiting index and leaf nodes, as well as data entries. Node and data information can be obtained using the INode

and IData interfaces (both extend IEntry). Examples of using this interface include visualizing a query, counting the number of leaf or index nodes visited for a specific query, throwing alerts when a specific spatial region is accessed, etc.

The queryStrategy method provides the ability to design more sophisticated queries. It uses the IQueryStrategy interface as a callback that is called continuously until no more entries are requested. It can be used to implement custom query algorithms (based on the strategy pattern [gamma94]).

A data entry can be inserted using the insertData method. The insertion function will convert any shape into an internal representation depending on the index. Every inserted object should be assigned an ID (called object identifier) that will allow updating, deleting and reporting the object. It is the responsibility of the caller to provide the index with IDs (unique or not). Also, a byte array can be associated with an entry. The byte arrays are stored along with the spatial information inside the leaf nodes. Clustered indices can be supported in that way. The byte array can also by null (in which case the length field should be zero), and no extra space should be used per node.

A data entry can be deleted using the deleteData method. The object shape and ID should be provided. Spatial indices cluster objects according to spatial characteristics and not IDs. Hence, the shape is essential for locating and deleting an entry.

Useful statistics are provided through the IStatistics interface and the getStatistics method.

Method getIndexProperties returns a PropertySet with all useful index properties like dimensionality etc.

A NodeCommand interface is provided for customizing Node operations. Using the addWriteNodeCommand, addReadNodeCommand and addDeleteNodeCommand methods, custom command objects are added in listener lists and get executed after the corresponding operations.

The isIndexValid method performs internal checks for testing the integrity of a structure. It is used for debugging purposes.

When a new index is created a unique index ID should be assigned to it, that will be used when reloading the index from persistent storage. This index ID should be returned as an IndexIdentifier property in the instance of the PropsertySet that was used for constructing the index instance. Using index IDs, multiple indices can be stored in the same storage manager. It is the users responsibility to manager the index IDs. Associating the wrong index ID with the wrong storage manager or index type has undefined results.

2.4 The RTree Package

The RTree index [guttman84] is a balanced tree structure that consists of index nodes, leaf nodes and data. Every node (leaf and index) has a fixed capacity of entries, (the node capacity) chosen at index creation An RTree abstracts the data with their Minimum Bounding Region (MBR) and clusters these MBRs according to various heuristics in the leaf nodes. Queries are evaluated from the root of the tree down the leaves. Since the index is balanced nodes can be under full. They cannot be empty though. A fill factor specifies the minimum number of entries allowed in any node. The fill factor is usually close to 70%.

RTree creation involves:

1. Deciding if the index will be internal or external memory and selecting the appropriate storage manager.
2. Choosing the index and leaf capacity (also known as fan-out).
3. Choosing the fill factor (from 1% to 99% of the node capacity).
4. Choosing the dimensionality of the data.
5. Choosing the insert/update policy (the RTree variant).

If an already stored RTree is being reloaded for reuse, only the index ID needs to be supplied during construction. In that case, some options cannot be modified. These include: the index and leaf capacity, the fill factor and the

dimensionality. Note here, that the RTree variant can actually be modified. The variant affects only when and how splitting occurs, and thus can be changed at any time.

An initialization PropertySet is used for setting the above options, complying with the following property strings:

Property	Type	Description
IndexIdentifier	VT_LONG	If specified an existing index will be opened from the supplied storage manager with the given index id. Behavior is unspecified if the index id or the storage manager are incorrect.
Dimension	VT_ULONG	Dimensionality of the data that will be inserted.
IndexCapacity	VT_ULONG	The index node capacity. Default is 100.
LeafCapactiy	VT_ULONG	The leaf node capacity. Default is 100.
FillFactor	VT_DOUBLE	The fill factor. Default is 70%
TreeVariant	VT_LONG	Can be one of Linear, Quadratic or Rstar. Default is Rstar
NearMinimumOverlapFactor	VT_ULONG	Default is 32.
SplitDistributionFactor	VT_DOUBLE	Default is 0.4
ReinsertFactor	VT_DOUBLE	Default is 0.3
EnsureTightMBRs	VT_BOOL	Default is true
IndexPoolCapacity	VT_LONG	Default is 100
LeafPoolCapacity	VT_LONG	Default is 100
RegionPoolCapacity	VT_LONG	Default is 1000
PointPoolCapacity	VT_LONG	Default is 500

2.5 Performance

Dataset size, data density, etc. have nothing to do with capacity and page size. What you are trying to achieve is fast reads from the disk and take advantage of disk buffering and prefetching.

Normally, you select the page size to be equal to the disk page size (depends on how you format the drive). Then you choose the node capacity to be enough to fill the whole page (including data entries if you have any).

There is a tradeoff though in making node capacity too large. The larger the capacity, the longer the "for loops" for inserting, deleting, locating node entries take (CPU time). On the other hand, the larger the capacity the shorter the tree becomes, reducing the number of random IOs to reach the leaves. Hence, you might want to fit multiple nodes (of smaller capacity) inside a single page to balance I/O and CPU cost, in case your disk page size is too large.

Finally, if you have enough buffer space to fit most of the index nodes in main memory, then large capacities do not make too much sense, because the height of the tree does not matter any more. Of course, the smaller the capacity, the larger the number of leaf nodes you will have to retrieve from disk for range queries (point queries and nearest neighbor queries will not suffer that much). So very small capacities hurt as well.

There is another issue when trying to fit multiple nodes per page: Leftover space. You might have leftover space due to data entries that do not have a fixed size. Unfortunately, in that case, leftover space per page is lost, negatively

impacting space usage.

Selecting the right page size is easy; make it equal to the disk page size. Selecting the right node capacity is an art…

CHAPTER

THREE

REFERENCES

[guttman84] "R-Trees: A Dynamic Index Structure for Spatial Searching" Antonin Guttman, Proc. 1984 ACM-SIGMOD Conference on Management of Data (1985), 47-57.

[gamma94] "Design Patterns: Elements of Reusable Object-Oriented Software" Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Addison Wesley. October 1994.

Author Marios Hadjieleftheriou

Contact mhadji@gmail.com

Revision 1.8.5

Date 11/01/2014

CHAPTER

FOUR

INTRODUCTION

4.1 Library Goals

The purpose of this library is to provide:

1. An extensible framework that will support robust spatial indexing methods.
2. Support for sophisticated spatial queries. Range, point location, nearest neighbor and k-nearest neighbor as well as parametric queries (defined by spatial constraints) should be easy to deploy and run.
3. Easy to use interfaces for inserting, deleting and updating information.
4. Wide variety of customization capabilities. Basic index and storage characteristics like the page size, node capacity, minimum fan-out, splitting algorithm, etc. should be easy to customize.
5. Index persistence. Internal memory and external memory structures should be supported. Clustered and non-clustered indices should be easy to be persisted.

4.2 Features

- Generic main memory and disk based storage managers.
- R*-tree index (also supports linear and quadratic splitting).
- MVR-tree index (a.k.a. PPR-tree).
- TPR-tree index.
- Advanced query capabilities, using Strategy and Visitor patterns.
- Arbitrary shaped range queries, by defining generic geometry interfaces.
- Large parameterization capabilities, including dimensionality, fill factor, node capacity, etc.
- STR packing / bulk loading.

4.3 Warnings

- The library is not thread-safe, even for seemingly read-only operations. Queries and updates must be run from within mutexes.

4.4 Licensing History

Note: libspatialindex changed from a LGPL to a MIT license as of the 1.8.0 release. For most situations, this should have no impact on the library's use, but it should open it up for usage in situations that otherwise might have been problematic. Versions of libspatialindex prior to 1.8.0 were licensed LGPL 2.0, with the license description on this file. The codebase has been been updated, with licensing information replaced in headers and source files, to use the MIT license as of the 1.8.0+ release.

This change was made to support the inclusion of software depending on libspatialindex in static linking-only environments such as embedded systems and Apple's iOS. libspatialindex versions prior to 1.8.0 will continue to live on as LGPL software, and developers can continue to contribute to them under terms of that license, but the main development effort, and ongoing maintenance, releases, and bug applications, will move forward using the new MIT license at http://github.com/libspatialindex/libspatialindex

4.5 License (MIT)

```
Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated documentation files (the "Software"),
to deal in the Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

CHAPTER

FIVE

DOWNLOAD

5.1 Current Release (MIT)

- **2014-09-29**
  - spatialindex-src-1.8.5-src.tar.gz (md5)
  - spatialindex-src-1.8.5-src.tar.bz2 (md5)

5.1.1 Windows Builds

Windows builds are provided for convenience. The full matrix might not be complete, and you will have to compile yourself using your favorite compiler configuration and cmake if something is missing.

- libspatialindex-1.8.5-win-msvc-2013-x64.zip (md5)

5.2 Past Release (MIT)

- **2012-12-13**
  - spatialindex-src-1.8.0-src.tar.gz (md5)
  - spatialindex-src-1.8.0-src.tar.bz2 (md5)

5.2.1 Windows Builds

Windows builds are provided for convenience. The full matrix might not be complete, and you will have to compile yourself using your favorite compiler configuration and cmake if something is missing.

- libspatialindex-1.8.0-win-msvc-2008-x64-x32.zip (md5)
- libspatialindex-1.8.0-win-msvc-2010-x64-x32.zip (md5)

5.3 Past Release(s) (LGPL)

- **2011-12-12**
  - spatialindex-src-1.7.1-src.tar.gz (md5)
  - spatialindex-src-1.7.1-src.tar.bz2 (md5)

- Release Notes