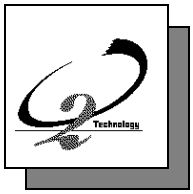


O_2 Spatial
User Manual

Release 5.0 - April 1998

Who should read this manual

This manual presents the O_2 spatial indexing module. This module allows you to develop efficient applications managing geographical information. The manual describes how to create, delete, update and query a spatial index associated with a collection of persistent objects. The O_2 spatial index module provides both high-level C++ and low-level O_2Engine API programming tools for spatial indexing. This manual contains a comprehensive list of O_2Spatial methods and commands.

Other documents available are outlined, click below.

See O2 Documentation set.

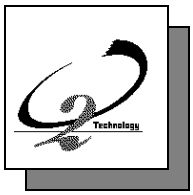


TABLE OF CONTENTS

This manual is divided into the following chapters:

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

1 Introduction

The O_2Spatial module provides spatial indexing and search capabilities for any collection of persistent objects in your O_2 database.

This chapter presents an overview of the O_2 system, an introduction to spatial indexing technology, and a brief overview of O_2Spatial.

To effectively use this manual, you should have some knowledge of O_2, its C++ ODMG binding interface and of OQL.

1 Introduction

1.1 System overview

The system architecture of O_2 is illustrated in Figure 1.1.

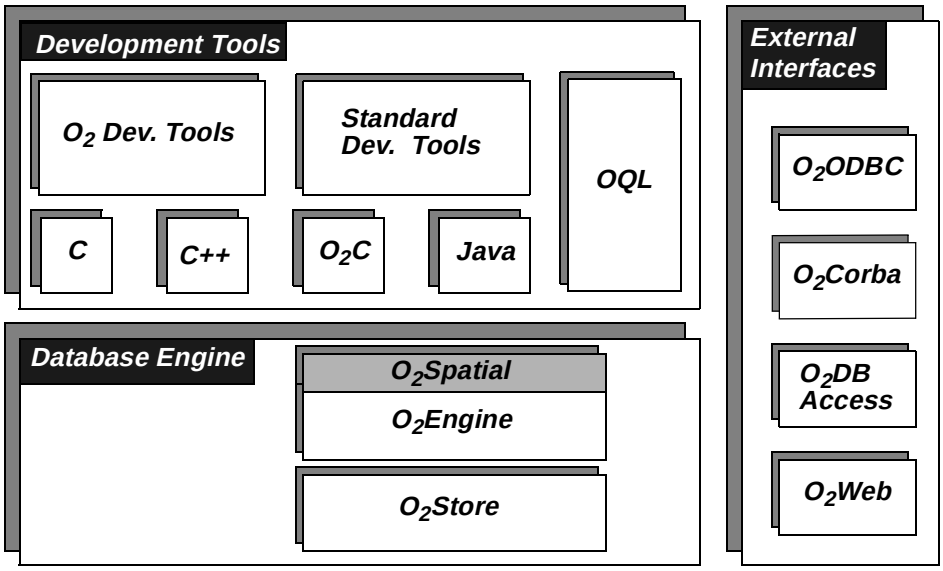


Figure 1.1: O_2 System Architecture

The O_2 system can be viewed as consisting of three components. The *Database Engine* provides all the features of a Database system and an object-oriented system. This engine is accessed with *Development Tools*, such as various programming languages, O_2 development tools and any standard development tool. Numerous *External Interfaces* are provided. All encompassing, O_2 is a versatile, portable, distributed, high-performance dynamic object-oriented database system.

Database Engine:

- O_2Store — The database management system provides low level facilities, through O_2Store API, to access and manage a database: disk volumes, files, records, indices and transactions.
- O_2Engine — The object database engine provides direct control of schemas, classes, objects and transactions, through O_2Engine API. It provides full text indexing and search capabilities with O_2Search and spatial indexing and retrieval capabilities with O_2Spatial. It includes a Notification manager for informing other clients connected to the same O_2 server that an event has occurred, a Version manager for handling multiple object versions and a Replication API for synchronizing multiple copies of an O2 system.

Programming Languages:

O_2 objects may be created and managed using the following programming languages, utilizing all the features available with O_2 (persistence, collection management, transaction management, OQL queries, etc.)

- C — O_2 functions can be invoked by C programs.
- C++ — ODMG compliant C++ binding.
- Java — ODMG compliant Java binding.
- O_2C — A powerful and elegant object-oriented fourth generation language specialized for easy development of object database applications.
- OQL — ODMG standard, easy-to-use SQL-like object query language with special features for dealing with complex O_2 objects and methods.

O_2 Development Tools:

- O_2Graph — Create, modify and edit any type of object graph.
- O_2Look — Design and develop graphical user interfaces, provides interactive manipulation of complex and multimedia objects.
- O_2Kit — Library of predefined classes and methods for faster development of user applications.
- O_2Tools — Complete graphical programming environment to design and develop O_2 database applications.

Standard Development Tools:

All standard programming languages can be used with standard environments (e.g. Visual C++, Sun Sparcworks).

External Interfaces:

- O_2Corba — Create an O_2/ Orbix server to access an O_2 database with CORBA.
- O_2DBAccess — Connect O_2 applications to relational databases on remote hosts and invoke SQL statements.
- O_2ODBC — Connect remote ODBC client applications to O_2 databases.
- O_2Web — Create an O_2 World Wide Web server to access an O_2 database through the internet network.

1.2 O_2 : Efficient support for Geographical Info Systems

The growth of computational power and memory capacities with the proliferation of personal computers has brought about the development of complex information systems.

Different forms and types of information can be represented as spatial data, such as medical images, satellite images, geographical information, etc.

Geographical entities require a high level model to be represented in a computational form. A geographical entity is by nature likely to be very complex. For instance a town map is made up of many different items such as streets, houses, parks, which can be themselves divided up into smaller parts. Each item has coordinates, scales and properties on a map.

The object model is very appropriate to capture this complexity, because of the natural and efficient way in which it describes each part as an object as well as the combination of items as composite objects. Thus O_2 is the perfect system to store and organize geographical information.

In order to be useful, such a repository should enable you to retrieve as fast as possible the information which you are interested in. Because of the geographical nature of the queries, they are linked to the topological properties of the objects. Some examples of queries are : which cities are **near** Paris? Which objects does New York City **include**? What rivers **cross** this town? etc. All these geographical operators work using graphical coordinates, which may be of a planar, a three-dimensional nature, or higher dimension as is the case of a hyper model. To efficiently answer such types of queries you need data structures and quick-access paths, which can deal with the coordinates of such items.

This is precisely the reason why you use a spatial index. The spatial index plays the same role regarding “geo-referenced” objects (i.e. objects referenced to by their coordinates) as a standard O_2 index regarding the simple properties of some objects (objects referenced to by the value of a simple attribute).

Similar to standard indexes (see chapter 5 in the *System Administration Guide*), a spatial index maps a key to a set of objects. The only difference lies in the nature of the key used. For a standard index, a key is an atomic data (e.g. integer, string...), whereas for a spatial index a key is a multi-dimensional data (e.g. a pair of reals : x, y; or a triple of reals : x, y, z; or more). Such multidimensional keys are called **spatial keys**.

1.3 The O_2Spatial module

This document presents the O_2 spatial indexing module. This module enables you to develop efficient applications managing geographical information (utilizing spatial indexing). The module uses the O_2 ODMG C++ binding and the OQL query language.

The spatial indexing module enables a programmer to create, delete, update and query a spatial index associated with a collection of persistent geographical objects.

This module provides the C++ programmer with an access to the spatial indexing capabilities of the O_2 system.

This document describes how to use spatial indexing, including creating and deleting using the O_2Shell, updating using C++, and querying using OQL and C++.

The O_2 spatial index module can also be used through a lower level programming interface. Low-level programming tools are provided by the O_2 Spatial Index Engine API as well and described in Chapter 5, Spatial indexing with O_2Engine API.

1.4 Object storage and retrieval

The proliferation of personal computers and distributed servers, together with the distribution of different forms and types of information across the World Wide Web, has increased the demand for powerful information storage and retrieval tools.

Spatial indexing enable users to search a large number of geo-referenced objects using specific selection criteria. Early spatial indexing systems allowed only 2-D objects. Today, spatial indexing systems are capable of efficiently processing multidimensional objects and supporting user queries that address specific object structures.

O_2Spatial

O_2Spatial enables application programmers to add spatial indexing and search capabilities to any collection of persistent objects stored in an O_2 database.

An O_2 application using O_2Spatial can perform three functions:

- Index collections of persistent geo-referenced objects to make them efficiently searchable
- Search and retrieve objects that satisfy users' queries
- Maintain and optimize spatial indexes

O_2Spatial applications can use the following search techniques:

- Searching for objects that intersect with a particular spatial point.
- Searching for objects that intersect with a certain window.
- Searching for objects that are inside a certain window.
- Searching for objects that are within a certain distance from a particular spatial point.

An application program interface (API) enables you to integrate spatial index capabilities within your O_2 applications.

Users' perspective

From the user's perspective, the primary interface to a spatial index is the query language, which is used to search a collection of geo-referenced objects. To offer the best flexibility and ease of use, ODMG OQL has been enhanced with specific operators for spatial searches (spatial predicates like "**`intersect`**" and "**`contain`**"). The result of a query is a set of objects or rectangular regions that

satisfy the search criteria. The user can choose an object from the resulting list and view it with an appropriate tool.

A C++ interface is provided which enables the user to manage the index as a collection of (key, object) pairs. This indexing interface is used whenever you want to add, update or delete an object from the spatial index.

R-Quad Tree

O_2Spatial is a spatial indexing system capable of efficient storing and retrieval of geo-referenced objects. This system is based on a Rectangle-Quadrant (R-Quad) tree. The spatial index is maintained by the O_2Spatial engine while the objects are maintained in an O_2 collection.

R-Quad tree based indexing and retrieval employs R-tree representation to store and organize information, and Quad-tree representation to accelerate spatial-based searching.

The main features of a spatial index based on a R-Quad tree architecture include the following:

- powerful query language that supports point, window and proximity searches.
- fast maintenance when objects are added and deleted.
- efficient retrieval even for very large collections and very large query results.

Spatial representation

When using the O_2Spatial engine, the spatial index stores a geo-referenced object as a minimum rectangle (called *bounding box*) that encompasses the object for identifying the spatial location of the object. To index an object, you simply provide the object and its bounding box.

O_2Spatial C++ API

Two C++ classes make up the O_2Spatial API: **`d_GeoCollection`** and **`d_SpatialKey`**.

O_2Spatial indexes and searches instances of **`d_SpatialKey`**, which represent indexed object keys.

An application can use methods in the **`d_GeoCollection`** class to search and retrieve objects stored in a spatially indexed collections. These methods produce lists of pointers to objects that satisfy the search criterion.

1 Introduction

2 Setting Up O_2Spatial

Various O_2 Spatial related files are needed to use the Spatial index. This chapter presents how to set up your O_2 system in order to use O_2 Spatial.

This chapter presents how to set up your O_2 system to use O_2 Spatial.

The chapter is composed of the following sections :

2 Setting Up O2Spatial

O2HOME Files

When you purchase O_2, O_2 Spatial is available upon request. After installing O_2, you can find the following O_2 Spatial-related files in your O_2 home directory. These files are needed to build an O_2 Spatial application:

Include_File/

o2spatial_CC.hxx	**- file you must include to use the O_2 Spatial C++ interface**
o2_spatialindex.h	**- file which must be included to use the spatial index interface of the O_2 Engine API.**
o2_spatial.oql	**- file you need to initialize O_2 schemas where spatial queries will be performed through OQL.**

Libraries/

libo2si.so	**- O_2 Spatial interface**
libo2si_engine.so	**- standard O_2 Spatial indexing engine**

You must add both libraries at link time when building a C++ application. This is achieved automatically by O_2Makegen. You just have to indicate in the configuration file :

+UseSpatialIndex

Refer to the *ODMG C++ Binding Guide* and the *O_2 Makegen User Manual* which describe how to write and build a C++ application with O_2.

When using the standard O_2 Spatial indexing engine with the spatial index interface of the O_2 Engine API, you also need the following include files :

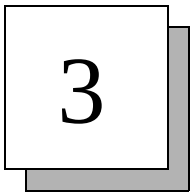
o2_SiArgList.hxx	**SiEllipse.hxx**
SiBoundingBox.hxx	**SiPolygon.hxx**
Si3DPoint.hxx	**SiPolyline.hxx**

Schema initialization

If you want to use spatial operators embedded in OQL queries, you must initialize the O_2 schema in which you are developing your application. To do so run **o2dsa_shell** with your system and server name. Then use the **set schema** command and include the OQL source file as follows :

```
$ o2dsa_shell -system your_system -server your_server
Type your command and end with ^D.
set schema your_schema;
#"O2HOME/include/o2_spatial.oql"
^D
```

2 Setting Up O2Spatial



Using O_2Spatial

This chapter presents an example of a simple O_2Spatial application. The example introduces the main components of the O_2Spatial architecture and shows how the O_2Spatial API can be used to build O_2 applications that make use of spatial indexing (SI) and retrieval capabilities. This example shows how to manage a collection of geographical objects, create a spatial index for this collection and carry out geographical queries. The geographical objects are ellipses whose definition is given by the class appearing on the next page.

The complete description of the API is in chapter 4.

The geographical object class GeoEllipse is defined as follows :

```
class GeoEllipse
{
   public:
      // Constructor
      GeoEllipse (
           int center_x, int center_y, int radius_x, int radius_y);

       // Specific methods
       void moveCenter (int newXCenter, int newYCenter);
      void stretch (int newXRadius, int newYRadius);

       // Spatial key of this ellipse
       d_SpatialKey2D* getSpatialKey () const;


   // Private attributes.
   private:
      int  xCenter;
      int  yCenter;
      int  xRadius;
      int  yRadius;
};
```

Note that a GeoEllipse object is defined by its center and lengths of its two extremum axes. Such representation is not possible for the O_2 Spatial index, as this sort of index can deal only with rectangles. Thus the class also needs the **`getSpatialKey`** method, which returns the coordinates of the minimum rectangle encompassing an ellipse. In the spatial index only such data (of **`SpatialKey2D`** type) will be inserted.

3.1 Spatial index creation and deletion

The creation and deletion commands

An O_2 spatial index is created and deleted through the O_2 database administration shell, like any other O_2 index (see the *O_2 System Administration Guide*). The O_2DBA shell commands are the following:

```
create spatial index <collection-name>
             [options "options-file-name"];
delete spatial index <collection-name>;
```

The **`create spatial index`** command creates a default spatial index structure associated with a named collection of persistent objects. The user then populates this index with the appropriate keys. The spatial index uses various option parameters, which are detailed below.

The **`delete spatial index`** command deletes the spatial index associated to a named collection of persistent objects.

The current spatial indexing system only allows a single spatial index for a given collection. Other types of indexes are of course allowed with no limitations.

The following example defines a collection of ellipses, each of which being represented by an instance of the C++ Class **`GeoEllipse`**. A spatial index is created for this collection. Note that both **`GeoEllipse`** and **`d_Set<d_Ref<GeoEllipse>>`** have been previously imported (see *C++ Binding Guide*).

```
constant name TheEllipses : o2_set_GeoEllipse;
create spatial index TheEllipses;
```

Spatial index options

The O_2 spatial index module uses a generic indexing algorithm based on *R-Quadtrees*. Each O_2 spatial index is initialized with default parameter values which may be customized as needed. For example, the dimensions of the keys and indexing space boundaries may be set for each spatial index.

There are two ways to set parameters for a spatial index :

- Either using a configuration file, which is specified when the index is created with **`o2dba_shell`** (in the options argument of the **`create spatial index`** command). This configuration file specifies a value for each option you want to customize, by providing a list of *option=value* (each on a separate line), where **option** is the option name and **value** is its value for the parameter;

- Or using a low-level interface described in Chapter 5, Spatial Indexing with O2Engine API .

The following options can be specified for the O_2 spatial index (the configuration file is not case-sensitive):

Option	Type	Description	Default value
IndexType	string	Name of the spatial index algorithm. Must be “RQuadTree” to use the O_2Spatial indexing algorithm.	RQuadTree
Dimension	integer	Number of dimensions in the indexed space.	2
KeyCoordType	string	Type used for spatial key coordinates, either "Integer16" or "Integer32".	Integer32

For each dimension *d* (i.e. for each axis of the indexed space), the following options can be specified:

Option	Type	Description	Default value
D*d*_Min	KeyCoord-Type value	Minimum coordinate value for the given dimension *d*.	KeyCoordType minimum value.
D*d*_Max	KeyCoordType value	Maximum coordinate value for the given dimension *d*.	KeyCoordType maximum value.

Other options are available for optimization of the spatial index (see section 3.3 Spatial Index tuning).

Example :

```
Dimension                    = 2
KeyCoordType                 = Integer32
D1_Min                       = 0
D1_Max                       = 10000
D2_Min                       = -100
D2_Max                       = 1000
```

3.2 C++ Interface to a spatial index

Contrary to a standard index O_2 does not automatically manage the spatial index, but it provides a programmative interface to populate the index.

Although in a future O_2 version, spatial indexes might be automatically maintained whenever possible, this process is still not possible sometimes, as a clear definition of a valid spatial key for a particular geographical object lacks.

Most of the time the application only knows how to approximate a geographical object by a rectangle. For instance, in our example, the **`getSpatialKey`** method only is able to compute the rectangle encompassing an ellipse.

It is thus generally impossible to determine when a spatial key has been changed, and on performance grounds we prefer not to systematically call a method, but rather rely on spatial keys directly passed by the programmer to access the spatial index.

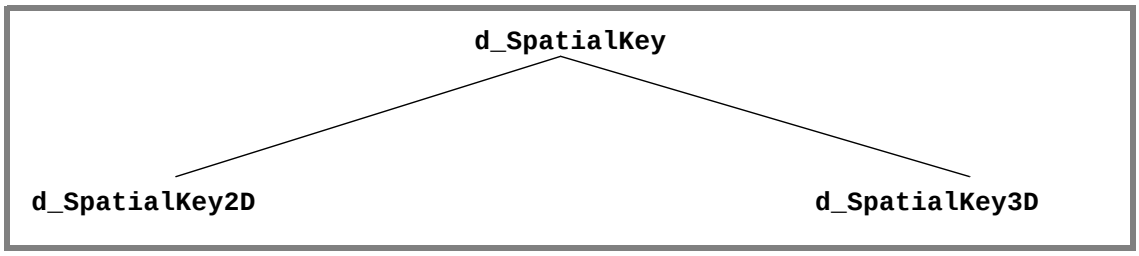
The O_2 system enables the C++ programmer to manipulate a spatial index as a collection of (key, value) pairs, such as the standard **`d_Dictionary`** ODMG class. Like this standard class the O_2 system uses the ODMG **`d_Association`** structure to associate a key and its value.

```
template <class K, class V>
class d_Association
{
public:
  K key;
  V value;
  d_Association();
  d_Association(const K& K, const V& V);
};
```

The spatial index module introduces the **d_GeoCollection**, which is the C++ interface for accessing the O_2 spatial indexes.

Like the standard **d_Collection** class, the **d_GeoCollection** class expects only one template parameter, which is the indexed object class. The indexing key class is assumed to be a subclass of the new abstract **d_SpatialKey** class.

For convenience, two concrete subclasses **d_SpatialKey2D** and **d_SpatialKey3D**, which inherit from the abstract **d_SpatialKey** class are also defined.



3.3 Spatial Index Constructor

To access the spatial index created in section 3.1 in C++, you have to use the constructor in combination with a name. By convention, this name is the same as that of the indexed collection.

```
d_GeoCollection<d_Ref<GeoEllipse> > Sindex("TheEllipses");
```

You can now populate the index with spatial key/GeoEllipse couples. At that point it is up to the application to populate or not the collection itself (the **o2_set_GeoEllipse** of the example). This is actually not compulsory, since no automatic maintenance is carried out. The index may be enough, as it represents a collection of the same objects.

3.4 Spatial Index update

Insertion and removal methods

The following methods are used to insert and remove spatial key/object couples to/from the spatial index :

```
void d_GeoCollection<T>::insert_element
    (const d_Association <d_SpatialKey*, T>& pair);
void d_GeoCollection<T>::remove_element
    (const d_Association <d_SpatialKey*, T>& pair);
```

Indexing or deleting an array of pairs is more efficient than indexing or deleting them individually, therefore two additional methods which manipulate arrays of pairs are provided:

.

```
void d_GeoCollection<T>::insert_elements
(const d_Array <d_Association <d_SpatialKey*, T>>& pairs);
void d_GeoCollection<T>::remove_elements
(const d_Array <d_Association <d_SpatialKey*, T>>& pairs);
```

The following example shows how a geo-referenced object can be added to the index.

```
d_Set<d_Ref<GeoEllipse> > Ellipses("TheEllipses");
d_GeoCollection<d_Ref<GeoEllipse> > Sindex("TheEllipses");
d_Association <d_SpatialKey*, d_Ref <GeoEllipse> >   pair;
d_Transaction transaction;

transaction.begin ();

// Set parameters for a new ellipse object.
int  center_x, center_y;
int  radius_x, radius_y;

cout << "Center:" << endl;
center_x = enter_number ("x=", 0, 32767);
center_y = enter_number ("y=", 0, 32767);
cout << "Radius:" << endl;
radius_x = enter_number ("x=", 0, 32767);
radius_y = enter_number ("y=", 0, 32767);

pair.value = new (database) GeoEllipse (
                            center_x, center_y, radius_x, radius_y);
pair.key = pair.value->getSpatialKey ();

// Insert the new object into the bag and the geo-collection.
Sindex.insert_element (pair);
Ellipses.insert_element (pair.value);

transaction.validate ();
```

Update method

If a spatialKey/Object association is modified you must update the O_2 system by calling the following method:

```
void d_GeoCollection<T>::update_element
    (const d_Association <d_SpatialKey*, T>& old_pair,
    const d_Association <d_SpatialKey*, T>& new_pair);
```

Note that you must call the **update** method for each modified object of the collection, by giving the values before update as well as the new values.

The complete C++ definition of the **d_GeoCollection** class and other classes, which are specific to the spatial index C++ interface, is given in Chapter 4.

3.5 Spatial Index tuning

The O_2 spatial index options

O_2 spatial indexes are maintained as *R-QuadTrees*. Several options enable these trees to be adapted and optimized to the user needs, for instance by allowing an *R-QuadTree* to be more balanced than the usual fixed *QuadTree*.

Firstly the number of dimensions used to describe a point in the indexed space is by default two. For a larger number, it must be declared in the option *Dimension* (up to O_2 page size/*KeyCoordType* size).

With the key coordinate type option *KeyCoordType*, you can reduce the size of the spatial index, by using a smaller key size (**Integer16**). Such a coarse-grid representation may be sufficient for certain applications. For a fine-grid use the default **Integer32** format.

Specific coordinate boundaries can be defined with the “D*d*_Min” and “D*d*_Max” options.

The full list of the O_2 spatial index tuning options are the following:

Option	Type	Description	Default value
IndexType	string	Name of the spatial index algorithm. Must be “RQuadTree” to use the O_2Spatial indexing algorithm.	RQuadTree
Dimension	integer	Number of coordinates of the indexed space.	2
OnlyPoints	boolean	True if indexed objects are always points.	False
KeyCoordType	string	Type used for spatial key coordinates. Must be "Integer16" or "Integer32".	Integer32
D*d*_Min	KeyCoordType value	Minimum coordinate value for the given dimension *d* (used for internal coordinates scaling optimization).	KeyCoordType minimum value
D*d*_Max	KeyCoordType value	Maximum coordinate value for the given dimension *d* (used for internal coordinates scaling optimization).	KeyCoordType maximum value
D*d*_Min RegionSize	integer	Minimum size of an indexed region (in KeyCoordType-based coordinates) for dimension *d*. Such a region will not be segmented more	2

3.6 Querying

Types of queries

A spatial index allows a user to run queries on geo-referenced data, and defines spatial predicates to do so. Four types of queries are supported :

- The pointing predicate (called *contain*), where all the objects that include a given point are retrieved,
- The windowing predicate (called *intersect*), where all the objects that touch a given area are retrieved (they can project beyond the area boundaries),
- The strict windowing predicate (called *inside*), where all the objects that are strictly within a given area are retrieved (i.e. none may project beyond the area boundaries),

- The closeness predicate (called *close_to*), where all the objects that are at a given distance from a given point or a given area are retrieved (they can project beyond the given distance).

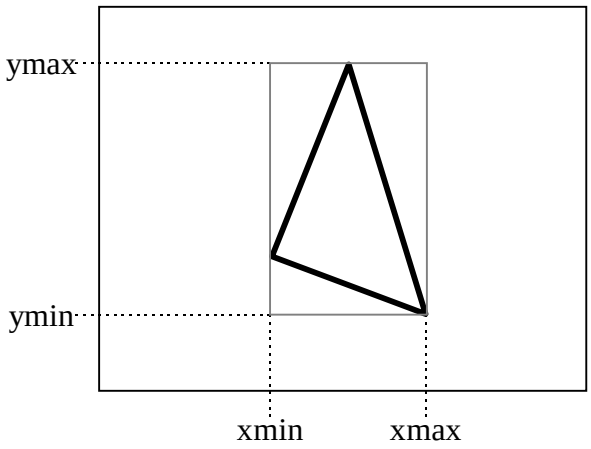
Graphical examples for each of these predicates are given in a following section.

Each spatial predicate is applied to bounding boxes (see figure below).

O_2 Spatial keys

The geographical position of each object is stored in the index. The key type used by the O_2 spatial indexing module is the minimum rectangle containing the indexed geographical objects, called the *bounding box*.

The bounding box principle is illustrated in the figure below:



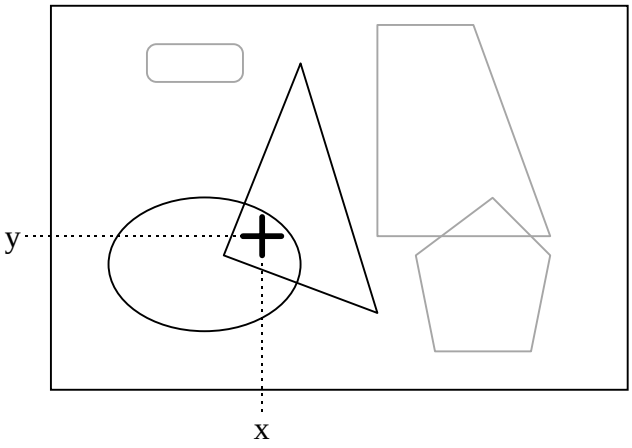
Here a geo-referenced object (the triangle) is drawn in black, and its bounding box is drawn in thin gray. When a bounding box has a surface area equal to zero (i.e. where *xmin = xmax* and *ymin = ymax*), it is considered to be a point and *xmax* and *ymax* may be omitted.

Spatial query examples

The following examples present some simple geometric queries.

- To retrieve all objects, which contain the point (**x**,**y**), from the spatial index **MyGeoObjects** collection, the appropriate query is:

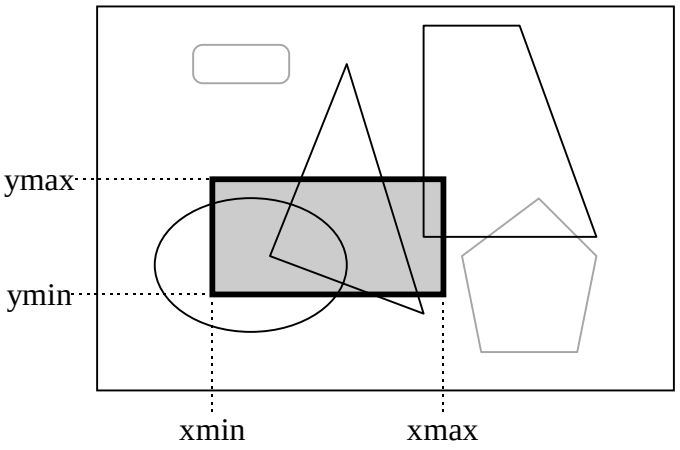
```
contain (MyGeoObjects, SpatialKey2D (x, y))
```



For this figure, the **contain** query returns the triangle and ellipse.

- Retrieve all the objects, which intersect with a specific window. This predicate can be used to zoom in or zoom out in accordance with the definition of the specific window, for example:

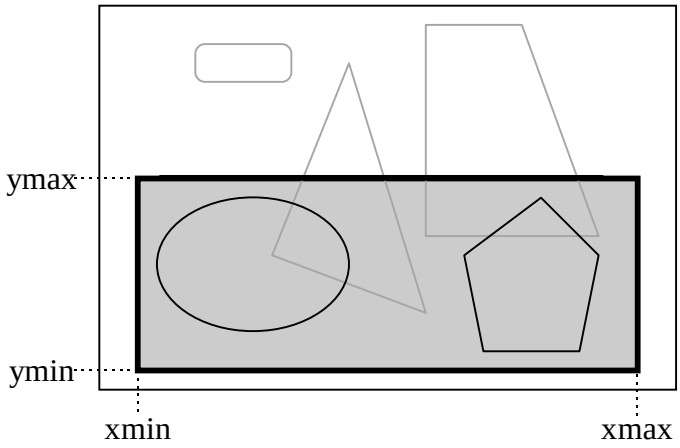
```
intersect (MyGeoObjects,
      SpatialKey2D (xmin, ymin, xmax, ymax))
```



For this figure, the **intersect** query returns the triangle, ellipse and trapezoid.

- Retrieve all the objects which are inside a specific window:

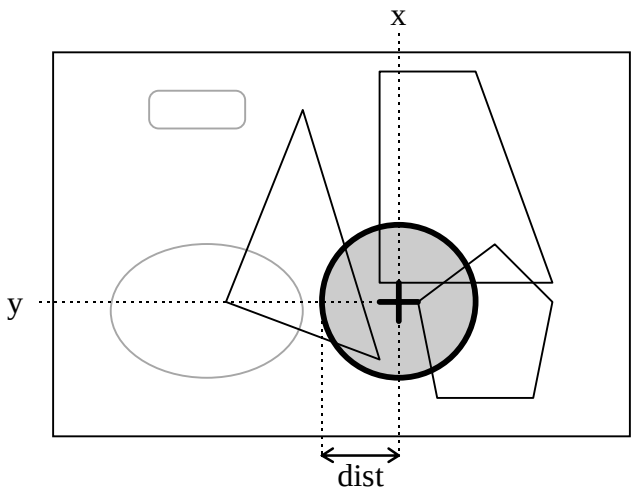
```
inside (MyGeoObjects,
       SpatialKey2D (xmin, ymin, xmax, ymax))
```



For this figure, the **inside** query returns the ellipse and pentagon, but neither the triangle nor the trapezoid, since they project beyond the window limits.

- Retrieve all the objects which are within a given distance of a point:

```
close_to (MyGeoObjects, SpatialKey2D (x, y), dist)
```



For this figure, the **close_to** query returns the triangle, trapezoid and pentagon.

Spatial predicate parameters

In the previous section, we presented how to use spatial predicates (in terms of points, rectangles and distances). This section describes the way these spatial predicates are used in the C++/OQL interface with spatial indexes.

The O_2 spatial index uses keys which are rectangles. Therefore such spatial keys can map directly onto objects which are rectangles or represented as rectangles. By extension, a point is considered as a zero-surface rectangle, hence a zero-surface spatial key can map an object which is a spatial point or represented as a point.

The notion of distance in multidimensional space makes sense only when the units of the considered axes are the same. In that case, the distance unit is in the given axis unit (remember that a distance is always a scalar and not a vector). If the axes do not use the same units, the closeness predicate will return incorrect results, and hence should not be used. Note that the closeness predicate does not verify whether the spatial units are equivalent or not.

How to query spatially

There are two ways of querying a spatial index :

- By calling a C++ method of the **`d_GeoCollection`** class
- By using OQL

Using object methods

For each spatial predicate, a **`d_GeoCollection`** method is defined. Each of these methods returns a **`d_Bag`**, which contains the found objects.

The following methods are defined in the **`d_GeoCollection`** class :

```
d_Bag<T> contain   (const d_SpatialKey& point);
d_Bag<T> intersect (const d_SpatialKey& window);
d_Bag<T> inside    (const d_SpatialKey& window);
d_Bag<T> close_to  (const d_SpatialKey& window,
                    d_Distance distance);
```

This object method querying approach offers a complete C++ solution to query a spatial index. Moreover, since these methods return a bag of objects, they can be applied in more complex queries, where both spatial and semantic predicates are used.

The following example uses the C++ querying approach:

```
{
   d_Bag <d_Ref <GeoEllipse> >  bagResult;
   int                          x, y;

   cout << "Query point:" << endl;
   x = enter_number ("x=", 0, 32767);
   y = enter_number ("y=", 0, 32767);

   bagResult = Sindex.contain (d_SpatialKey2D (x, y));
}
```

Using OQL predicates

For each spatial query type, a function callable by OQL is defined, as shown in the following :

```
geo_contain (coll : Object, point : SpatialKey): o2_bag;
geo_intersect (coll Object, window : d_SpatialKey): o2_bag;
geo_inside (coll : Object, window : d_SpatialKey): o2_bag;
geo_close_to  (coll : Object, window : d_SpatialKey,
                      distance : Distance): o2_bag;
```

These functions should be explicitly included (see Chapter 2) in each schema where these operators are to be used. Here, the `Distance` type is a named type for `double`.

The above functions are similar to the `d_GeoCollection` query methods. They return a bag of indexed objects, like the `d_GeoCollection` query methods.

Like the `d_GeoCollection` query methods, these functions can be applied in more complex queries, where both spatial and semantic predicates are used.

The following example uses the OQL querying approach :

```
{
    d_Query query ("geo_intersect (TheEllipses, $1)")
    d_Bag <d_Ref <GeoEllipse> >        bagResult;
    int                                x, y;

    cout <<"Query point:" << endl;
    x = enter_number ("x=", 0, 32767);
    y = enter_number ("y=", 0, 32767);

    d_SpatialKey key (x, y);

    query << &key;

    d_oql_execute (bagResult, query);

}
```

Examples

Get all the hotels at a distance less than 5 kilometres from Colorado Falls:

```
d_Query query (“geo_close_to (TheHotelsOfUSA, $1, $2)”);
query << ColoradoFalls->get position();
query << 5; // Distance in kilometers
```

Search a window (e.g. to zoom in) of coordinates (*xmin, ymin, xmax, ymax*) :

```
d_SpatialKey keyWindow (xmin, ymin, xmax, ymax);
d_Query query (“geo_intersect (MyGeoObjects, $1)”);
query << &keyWindow;
```

Get all the Sequoia forests which belong to the Yellowstone National Park and whose surface is greater than 1,000 square kilometers :

```
d_Query query (“select forest”
“from forest in (o2_bag_Forest) geo_intersect (SequoiaForests, $1)”
“where forest.surface >= $2”);
```

The two parameters are :

```
query << YellowstonePark->getSpatialKey();  // a bag of forests
query << 1000;                              // a surface.
```

If the intersect predicate is very selective this query runs fast. Otherwise if the surface is very selective this query cannot use the standard index on the attribute surface.

If you want to use both indexes[1] then you must rewrite the query as an intersection of the two result sets :

```
(o2_bag_Forest(select forest from forest in SequoiaForests
 where forest.surface >= $2))
intersect
((o2_bag_Forest) geo_intersect (SequoiaForests, $1))
```

1. Note : in a future version of OQL, the OQL optimizer will rewrite this automatically

4 O_2Spatial C++ API Reference

The O_2Spatial C++ API enables users to use O_2Spatial indexing and searching facilities for a collection of persistent objects.

When an error is detected, O_2Spatial throws a ODMG C++ exception containing an error code and description (see the class d_Error in the *ODMG C++ Reference Manual*).

This chapter describes the public interface of the C++ classes that make up the O_2Spatial API. The following classes are included:

- d_SpatialKey
- d_SpatialKey2D
- d_SpatialKey3D
- d_GeoCollection

4.1 Class definitions

The public interface of the class is defined as follows:

```
typedef double d_Distance;

class d_SpatialKey : public d_Object
{
public:
   d_SpatialKey ();
   d_SpatialKey (const d_SpatialKey& key);
   virtual ~d_SpatialKey ();
   virtual d_SpatialKey* clone () const = 0;

   // Dimension of the indexed space.
   virtual int dimension () const = 0;
   // O2 key class name.
   virtual const char* o2_get_name () const = 0;
   // Internal spatial key.
   virtual const void* getInternalKey () const = 0;
   // Internal spatial key size.
   virtual int getSize () const = 0;
   // Checks whether the key is a point.
   virtual int isAPoint () const = 0;
   // Enlarges the key.
   virtual void enlarge (d_Distance distance) = 0;
   // Assignment operator.
   virtual const d_SpatialKey& operator= (const d_SpatialKey&);
   // Key union operator.
   virtual const d_SpatialKey& operator+= (const d_SpatialKey&);

};
```

```
class d_SpatialKey2D : public d_SpatialKey
{
public:
   d_SpatialKey2D (long x, long y);
   d_SpatialKey2D (long xmin, long ymin,
                   long xmax, long ymax);
   d_SpatialKey2D (const d_SpatialKey2D& key);
   virtual ~d_SpatialKey2D ();
   virtual d_SpatialKey* clone () const;

   // Return 2.
   virtual int dimension () const;
   virtual const char* o2_get_name () const;
   virtual const void* getInternalKey () const;
   virtual int getSize () const;
   virtual int isAPoint () const;
   virtual void enlarge (d_Distance distance);
   virtual const d_SpatialKey& operator= (const d_SpatialKey&);
   virtual const d_SpatialKey& operator+= (const d_SpatialKey&);

   // Return the key coordinates.
   long xmin () const;
   long xmax () const;
   long ymin () const;
   long ymax () const;
};
```

```
class d_SpatialKey3D : public d_SpatialKey
{
public:
   d_SpatialKey3D (long x, long y, long z);
  d_SpatialKey3D (long xmin, long ymin, long zmin, long xmax, long ymax,
                  long zmax);
   d_SpatialKey3D (const d_SpatialKey3D& key);
   virtual ~d_SpatialKey3D ();
   virtual d_SpatialKey* clone () const;

   // Return 3.
   virtual int dimension () const;
   virtual const char* o2_get_name () const;
   virtual const void* getInternalKey () const;
   virtual int getSize () const;
   virtual int isAPoint () const;
   virtual void enlarge (d_Distance distance);
   virtual const d_SpatialKey& operator= (const d_SpatialKey&);
   virtual const d_SpatialKey& operator+= (const d_SpatialKey&);

   // Return the key coordinates.
   long xmin () const;
   long xmax () const;
   long ymin () const;
   long ymax () const;
   long zmin () const;
   long zmax () const;
};
```

```
template <class T>
class d_GeoCollection
{
public:
   d_GeoCollection ();
   d_GeoCollection (const char* collection_name);
   d_GeoCollection (Handle hdCollection);
   d_GeoCollection (const d_GeoCollection<T>& coll);
  ~d_GeoCollection ();
   d_GeoCollection& operator= (const d_geoCollection<T>& coll);

   void insert_element (const d_Association <d_SpatialKey*, T>& pair);
   void remove_element (const d_Association <d_SpatialKey*, T>& pair);
   void update_element (const d_Association <d_SpatialKey*, T>& old_pair,
                        const d_Association <d_SpatialKey*, T>& new_pair);
   void insert_elements (const d_Array <d_Association <d_SpatialKey*,
                        T>>& pairs);
   void remove_elements (const d_Array <d_Association <d_SpatialKey*,
                        T>>& pairs);

   d_Bag<T> contain   (const d_SpatialKey& point);
   d_Bag<T> intersect (const d_SpatialKey& window);
   d_Bag<T> inside    (const d_SpatialKey& window);
   d_Bag<T> close_to  (const d_SpatialKey& window,
                       const d_Distance distance);
   };
```

4.2 Member functions descriptions

This section gives the full description of the following C++ member functions for spatial index management:

For d_GeoCollection :

- close_to
- contain
- d_GeoCollection (constructor)
- d_GeoCollection (constructor)
- insert_element
- insert_elements
- inside
- intersect
- operator=
- remove_element
- remove_elements
- update_element

For d_SpatialKey :

- clone
- d_SpatialKey2D (constructor)
- d_SpatialKey2D (constructor)
- d_SpatialKey3D (constructor)
- d_SpatialKey3D (constructor)
- dimension
- enlarge
- getInternalKey
- getSize
- isAPoint
- o2_get_name
- operator=
- operator+=

close_to

Summary Gets the objects, whose minimum rectangle is within the distance of the rectangle window (if the window size is equal to zero, it is considered as a point).

Syntax

```
d_Bag<T> d_GeoCollection<T>::close_to
(const d_SpatialKey& window, const d_Distance distance);
```

Arguments

window	Window (rectangle) to be searched.
distance	Distance added to the window which defines the search space.

Returns Returns a bag containing the objects found.

Error Throws a d_Error object (see the *ODMG C++ Reference Guide*) if the following error occurs :

- spatial key out of bounds (**d_Error::InvalidSpatialKey**).

contain

Summary Gets the objects, whose minimum rectangle contains the point.

Syntax

```
d_Bag<T> d_GeoCollection::contain
                (const d_SpatialKey& point);
```

Arguments **point** Zero-surface key to be tested.

Returns Returns a bag containing the objects found.

Error None.

d_GeoCollection (constructor)

Summary Opens a collection indexed with a spatial index.

Syntax `d_GeoCollection<T>::d_GeoCollection (Handle hdCollection);`

Arguments `hdCollection` Handle to the collection to be opened.

Returns None.

Error Throws a **`d_Error`** object (see the ODMG C++ Reference Guide) if one of the following errors occur :

- no database has been opened (**`d_Error::NoOpenedDatabase`**),
- no such named collection exists in the current database (**`d_Error::NoSuchNamedCollection`**),
- the collection is not indexed with a spatial index (**`d_Error::UnknownIndex`**).

d_GeoCollection (constructor)

Summary Opens a named collection indexed with a spatial index.

Syntax

```
d_GeoCollection<T>::d_GeoCollection
          (const char* CollectionName);
```

Arguments **`CollectionName`** Database name of the collection to be opened.

Returns None.

Error Throws a **`d_Error`** object if one of the following errors occurs:

- no database has been opened (**`d_Error::NoOpenedDatabase`**).
- no such named collection exists in the current database. (**`d_Error::NoSuchNamedCollection`**)
- the collection is not indexed with a spatial index. (**`d_Error::UnknownIndex`**)

insert_element

Summary Inserts an element into the spatial index.

Syntax
```
void d_GeoCollection<T>::insert_elements
  (const d_Association <d_SpatialKey*, T>& pair);
```

Arguments **pair** Pair to be inserted within the spatial index.

Returns None.

Error None.

insert_elements

Summary Inserts several elements into the spatial index.

Syntax

```
void d_GeoCollection<T>::insert_elements
(const d_Array <d_Association <d_SpatialKey*, T>>& pair);
```

Arguments **pair** Array pairs to be inserted within the spatial index.

Returns None.

Error None.

inside

Summary Gets the objects, whose minimum rectangle is strictly included in the rectangle window.

Syntax

```
d_Bag <T> d_GeoCollection<T>::inside
                (const d_SpatialKey& window);
```

Arguments **window** Window space to be searched.

Returns Returns a bag containing the objects found.

Error None.

intersect

Summary Gets the objects, whose minimum rectangle has a non-empty intersection with the rectangle window.

Syntax

```
d_Bag <T> d_GeoCollection<T>::intersect
                (const d_SpatialKey& window);
```

Arguments **window** Window space to be searched.

Returns Returns a bag containing the objects found.

Error None.

operator=

Summary Assigns a d_GeoCollection object to another.

Syntax
```
d_GeoCollection& d_GeoCollection<T>::operator=
          (const d_GeoCollection& coll);
```

Arguments Collection to be opened.

Returns None.

Error None.

remove_element

Summary Removes an element from the spatial index.

Syntax

```
void d_GeoCollection<T>::remove_elements
  (const d_Association <d_SpatialKey*, T>& pair);
```

Arguments **pair** Pair to be removed from the spatial index.

Returns None.

Error None.

remove_elements

Summary Removes several elements from the spatial index.

Syntax
```
void d_GeoCollection<T>::remove_elements
(const d_Array <d_Association <d_SpatialKey*, T>>& pair);
```

Arguments **pair** Array pairs to be removed from the spatial index.

Returns None.

Error None.

update_element

Summary Updates an element of the spatial index.

Syntax

```
void d_GeoCollection<T>::update_element
   (const d_Association <d_SpatialKey*, T>& old_pair
    const d_Association <d_SpatialKey*, T>& new_pair);
```

Arguments

old_pair	Pair to be updated within the spatial index
new_pair	New value of the pair to be updated.

Returns None.

Error None.

clone

Summary	Gets a copy of a spatial key, whatever its dimension.
Syntax	`d_SpatialKey* d_SpatialKey::clone () const;`
Arguments	None.
Returns	A pointer to the key copy of a nil pointer if any error occurred.
Error	If any copy error occurred, returns a nil pointer.

d_SpatialKey2D (constructor)

Summary Builds a zero-surface 2-D spatial key from the corresponding point coordinates.

Syntax `d_SpatialKey2D::d_SpatialKey2D (long x, long y);`

Arguments **x**, **y** New spatial key coordinates.

Returns None.

Error None.

.

d_SpatialKey2D (constructor)

Summary Builds a 2-D spatial key from the corresponding rectangle coordinates.

Syntax
```
d_SpatialKey2D::d_SpatialKey2D (long xmin, long ymin,
                                long xmax, long ymax);
```

Arguments **xmin**, **ymin**, **xmax**, **ymax** New spatial key coordinates.

Returns None.

Error None.

d_SpatialKey3D (constructor)

Summary Builds a zero-surface 3-D spatial key from the corresponding point coordinates .

Syntax `d_SpatialKey3D::d_SpatialKey3D (long x, long y, long z);`

Arguments **x**, **y**, **z** New spatial key coordinates.

Returns None.

Error None.

d_SpatialKey3D (constructor)

Summary Builds a 3-D spatial key from the corresponding rectangle coordinates.

Syntax
```
d_SpatialKey3D::d_SpatialKey3D (long xmin, long ymin,
      long zmin, long xmax, long ymax, long zmax);
```

Arguments **xmin**, **ymin**, **zmin**, **xmax**, **ymax**. **zmax** New spatial key coordinates.

Returns None.

Error None.

dimension

Summary Gets the indexed space dimension (two if two coordinates are required to describe a point, three if three coordinates are needed, and so on).

Syntax `virtual int d_SpatialKey::dimension () const;`

Arguments None.

Returns Returns the dimension as an integer.

Error None.

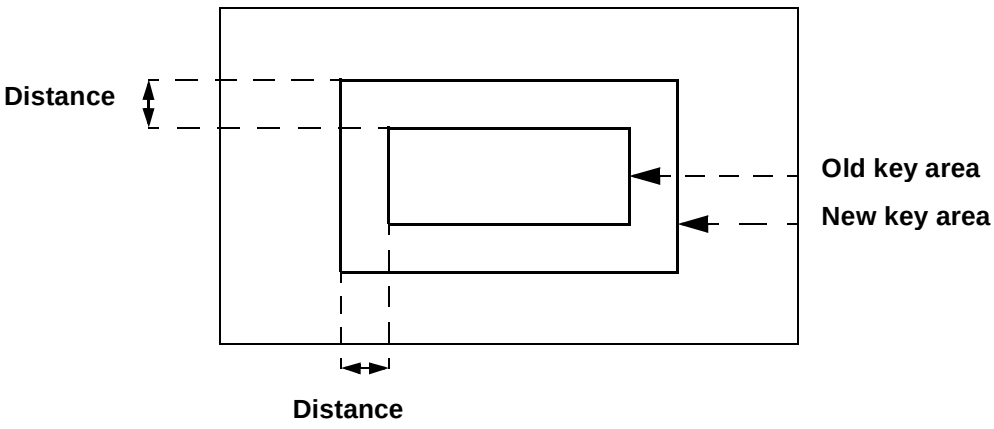
enlarge

Summary Resizes the spatial key.

Syntax `void d_SpatialKey::enlarge (d_Distance distance);`

Arguments The distance.

Returns Returns an enlarged key as shown in the following 2-D drawing :



Error None.

.

getInternalKey

Summary Gets the internal spatial key structure.

Syntax
```
const void* d_SpatialKey::getInternalKey () const;
```

Arguments None.

Returns Returns a pointer to the internal spatial key structure.

Error None.

getSize

Summary	Gets the internal spatial key structure size.
Syntax	`int d_SpatialKey::getSize () const;`
Arguments	None.
Returns	Returns the size in bytes.
Error	None.

isAPoint

Summary Checks whether the spatial key is a point.

Syntax
```
int d_SpatialKey::isAPoint () const;
```

Arguments None.

Returns Returns TRUE if the spatial key is a zero-surface rectangle, and FALSE otherwise.

Error None.

o2_get_name

Summary	Gets the key class name used by O_2 for its own type verification.
Syntax	`const char* d_SpatialKey::o2_get_name () const;`
Arguments	None.
Returns	Returns a string which contains the key class name.
Error	None.

operator=

Summary Assigns a key to another.

Syntax
```
d_SpatialKey& d_SpatialKey::operator=
(const d_SpatialKey& key) const;
```

Arguments The key to be assigned.

Returns None.

Error None.

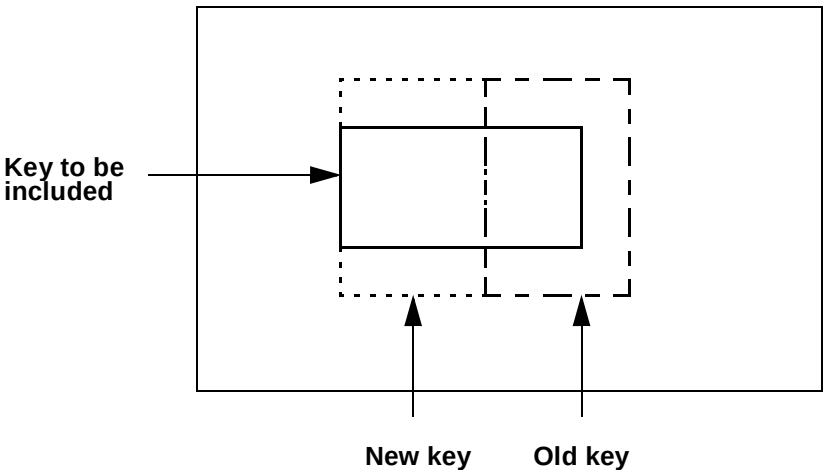
operator+=

Summary Enlarges a key with the area covered by another key.

Syntax
```
virtual const d_SpatialKey& d_SpatialKey::operator+=
(const d_SpatialKey&) const;
```

Arguments The key to be included.

Returns Returns an enlarged key as shown in the following 2-D drawing :



Error None.

4 O2Spatial C++ API Reference

5 Spatial Indexing with O_2Engine API

This chapter describes the use of spatial indexing with the O_2Engine API.

It contains the following sections:

5 Spatial Indexing with O2Engine API

5.1 Introduction

The O_2Engine API Spatial Index interface is made up of a set of functions that allow you to maintain and access a spatial index of an O_2 database.

The library of spatial index functions allows to create, remove, update and query the spatial index associated with a collection of O_2 objects.

5.2 O_2 Engine API : Spatial Index

The spatial index interface of the O_2Engine API defines a collection of operations available to the developer of O_2Engine applications for managing a spatial index.

The commands are divided into three categories of operations:

1. The operations on the spatial index :

- **o2x_spatial_index_create**
- **o2x_spatial_indexed**
- **o2x_spatial_index_delete**
- **o2x_spatial_index_reorganize**
- **o2x_spatial_index_get_info**
- **o2x_spatial_index_free_descriptor**
- **o2x_spatial_index_get_collection**

2. The operations on the elements of the spatial index :

- **o2x_spatial_index_insert_key**
- **o2x_spatial_index_group_insert_key**
- **o2x_spatial_index_delete_key**
- **o2x_spatial_index_group_delete_key**
- **o2x_spatial_index_replace_key**

3. The search operations on the spatial index :

- **o2x_spatial_index_scan_open**
- **o2x_spatial_index_scan_close**
- **o2x_spatial_index_scan_read**

o2x_spatial_index_create

Summary Create a spatial index associated with a collection of persistent objects.

Syntax

```
#include <o2.h>
#include <o2_error.h>
#include <o2_spatialindex.h>
int o2x_spatial_index_create( Handle colHd,
                              char * volumeName,
                              void *opaque);
```

Description Verifies in the spatial index catalogue that the collection indicated by the parameter **colHd** has not been previously associated with a spatial index.

Creates the O_2 Store file for the spatial index tree in the volume declared by the parameter **volumeName** or in the same volume of the collection, if **volumeName** is set to zero.

Initializes the spatial index as a function of the parameter **opaque**. When using the O_2 Spatial indexing engine, the parameter **opaque** is expected to be of type **o2_SiArgList***. This class enables the user to set all the spatial index parameters as (option, value) couples.

The following example shows how to build such an object :

```
o2_SiArgList argList;
argList.addArgument("IndexType", "RQuadTree");
argList.addArgument("KeyCoordType", "Integer32");
argList.addArgument("Dimension", 2);
```

The arguments, which must be set by using an **o2_SiArgList** object, include those described in section 3.5. The arguments are described below :

Option	Type	Description	Recommended value
IndexType	string	Name of the spatial index algorithm. Must be “RQuadTree” to use the O_2Spatial indexing algorithm.	RQuadTree
Dimension	integer	Number of coordinates of the indexed space.	2
OnlyPoints	boolean	True if indexed objects are always points.	False
KeyCoordType	string	Type used for spatial key coordinates. Must be "Integer16" or "Integer32".	Integer32
D*d*_Min	KeyCoordType value	Minimum coordinate value for the given dimension *d* (used for internal coordinates scaling optimization).	KeyCoordType minimum value
D*d*_Max	KeyCoordType value	Maximum coordinate value for the given dimension *d* (used for internal coordinates scaling optimization).	KeyCoordType maximum value
D*d*_Min RegionSize	integer	Minimum size of an indexed region (in KeyCoordType-based coordinates) for dimension *d*. Such a region will not be segmented more	2
MaxChildren	integer	Maximum number of subspaces generated each time a given indexed space needs to be segmented (used when a R-QuadTree node is saturated)	$2^{Dimension}$
Threshold	integer	Number of (key, values) couples in each R-QuadTree node. Must be greater or equal to 2, and less or equal than : - 167 if KeyCoordType= integer 32 Dimension = 2 - 126 if KeyCoordType=Integer 32 Dimension=3	Highest possible value (see description)

At this level of programming no default value is provided for these parameters. An appropriate value must then be set up for all of them.

Returns **O2_OK** if successful, error code otherwise.

Errors **O2E_NOTSUPPORTED**
The function is not implemented.

O2E_SIMALGO
Error in Spatial Index Module algorithm.

O2E_PARAMETER
The parameter **opaque** is incorrect.

O2E_INVALIDCOLLECTION
The handle **colHd** is invalid or is not a handle to a persistent collection.

O2E_IS_INDEXED
There is no spatial index for this collection.

O2E_NOSPACEONDISK
No more space on disk.

O2E_PERMISSIONDENIED
The handle **colHd** is valid, but no spatial index can be created on a collection of versionned objects.

O2E_ERROR
Internal error.

Sample

```
int create_spidx (Handle      hdCollection,
                  o2_Boolean  indexOnlyContainsPoints)
{
   o2_SiArgList   SIOptions;

   SIOptions.addArgument ("IndexType",    "RQuadTree");
   SIOptions.addArgument ("KeyCoordType", "Integer32");
   SIOptions.addArgument ("Dimension",              2);
   SIOptions.addArgument ("D1_Min",                 0);
   SIOptions.addArgument ("D2_Min",                 0);
   SIOptions.addArgument ("D1_Max",             40000);
   SIOptions.addArgument ("D2_Max",             40000);
   SIOptions.addArgument ("D1_MinRegionSize",       2);
   SIOptions.addArgument ("D2_MinRegionSize",       2);
   SIOptions.addArgument ("MaxChildren",            4);
   SIOptions.addArgument ("Threshold",            167);
   SIOptions.addArgument ("OnlyPoints",
indexOnlyContainsPoints);

  return o2x_spatial_index_create (hdCollection, 0, (void*)
&SIOptions);
}
```

o2x_spatial_indexed

Summary Returns a handle to the spatial index associated with a collection of persistent objects.

Syntax

```
#include <o2.h>
#include <o2_error.h>
#include <o2_spatialindex.h>
int o2x_spatial_indexed(  Handle colHd,
                          o2x_SpatialIndexDesc **idxDesc);
```

Description Searches for the spatial index associated with a collection identified by **colHd** in the spatial index catalogue.

If the corresponding entry exists, a handle for the spatial index is created and its value is assigned to the parameter **idxDesc**.

Otherwise the value NULL is assigned to the parameter **idxDesc**.

The spatial index descriptor created by the **o2x_spatial_indexed** function should always be deleted using the **o2x_spatial_index_free_descriptor** function.

Returns **O2_OK** if successful, error code otherwise.

Errors

O2E_PARAMETER
The parameter **colHd** or **idxDesc** is incorrect.

O2E_NOTINDEXED
There is no spatial index for this collection.

O2E_INVALID_COLLECTION
The handle **colHd** is invalid or is not a handle to a persistent collection.

o2x_spatial_index_delete

Summary Remove a spatial index associated with a collection of persistent objects.

Syntax

```
#include <o2_error.h>
#include <o2_spatialindex.h>
int o2x_spatial_index_delete(
                              o2x_SpatialIndexDesc *idxDesc);
```

Description Deletes the spatial index identified by the parameter **idxDesc**, and removes the corresponding entry in the spatial index catalogue.

The operation is not executed if there exists a **scan** operation running at the time of the call.

Returns **O2_OK** if successful, error code otherwise.

Errors

O2E_SIMALGO
Error in Spatial Index Module algorithm.

O2E_PARAMETER
The parameter **idxDesc** is incorrect

O2E_ILLEGALCALL
Function was called during a scan.

o2x_spatial_index_reorganize

Summary Reorganize the distribution of index nodes on a disk page.

Syntax

```
#include <o2_error.h>
#include <o2_spatialindex.h>
int o2x_spatial_index_reorganize(
                                o2x_SpatialIndexDesc *idxDesc);
```

Description The function **o2x_spatial_index_reorganize** gathers the spatial index nodes on a disk page so as to reduce the access time during a search of the index tree.

This function fails if a **scan** operation is active on the spatial index.

Returns **O2_OK** if successful, error code otherwise.

Errors

O2E_NOTSUPPORTED
The function is not implemented.

O2E_SIMALGO
Error in Spatial Index Module algorithm.

O2E_PARAMETER
The parameter **idxDesc** is incorrect.

O2E_ILLEGALCALL
Function was called during a scan.

O2E_NOSPACEONDISK
No more space on disk.

o2x_spatial_index_get_info

Summary Collects information on the spatial index.

Syntax

```
#include <o2_error.h>
#include <o2_spatialindex.h>
int o2x_spatial_index_get_info(
                            o2x_SpatialIndexDesc *idxDesc,
                              o2x_SpatialIndexInfo *idxInfo);
```

Description Provides the requested information for the structure **idxInfo**.

This function returns statistical information on the spatial index, according to the bit mask provided in the structure **idxInfo**.

The type **o2x_SpatialIndexInfo** is defined as follows:

```
typedef struct {
   unsigned int    mask;
   int             dimension;
   int             depth;
   int             entryCount;
   int             allNodeCount;
   int             nodeCount;
   double          nodeMeanFillFactor;
   int             leafCount;
   int             emptyLeafCount;
   double          leafMeanFillFactor;
} o2x_SpatialIndexInfo;
```

The values which can be used to set the bit mask **mask** are as follows :

```
O2X_SI_DIMENSION,
O2X_SI_DEPTH,
O2X_SI_ENTRYCNT,
O2X_SI_ALLNODECNT,
O2X_SI_NODEMFF,
O2X_SI_LEAFCNT,
O2X_SI_EMPTYLEAFCNT,
O2X_SI_LEAFMFF
```

- If **mask & O2X_SI_DIMENSION** is TRUE, **dimension** contains the dimension of the spatial index.
- If **mask & O2X_SI_DEPTH** is TRUE, **depth** contains the depth of the spatial index tree.

- If **mask & O2X_SI_ENTRYCNT** is TRUE, **entryCount** contains the number of entries (**key**, **element**) in the spatial index tree. Note that the number of spatial index entries may be greater than the number of objects really inserted in the index, since some objects may belong to several quadrants.

- If **mask & O2X_SI_ALLNODECNT** is TRUE, **allNodeCount** contains the total number of nodes in the spatial index tree.

- If **mask & O2X_SI_NODECNT** is TRUE, **nodeCount** contains the number of intermediary nodes in the spatial index tree including the root node.

- If **mask & O2X_SI_NODEMFF** is TRUE, **nodeMeanFillFactor** contains the percentage of occupied intermediary nodes in the spatial index tree.

- If **mask & O2X_SI_LEAFCNT** is TRUE, **leafCount** contains the number of leaf nodes in the spatial index tree.

- If **mask & O2X_SI_EMPTYLEAFCNT** is TRUE, **emptyLeafCount** contains the number of empty leaf nodes in the spatial index tree.

- If **mask & O2X_SI_LEAFMFF** is TRUE, **leafMeanFillFactor** contains the percentage of occupied leaf nodes in the spatial index tree.

  The percentage of occupied leaves is distinguished from the percentage of occupied intermediary nodes in that leaves and nodes can contain different maximum numbers of keys. Moreover, the notion of an empty leaf may not be significant for certain spatial index algorithms.

Returns **O2_OK** if successful, error code otherwise.

Errors

O2E_NOTSUPPORTED
The function is not implemented.

O2E_SIMALGO
Error in Spatial Index Module algorithm.

O2E_ILLEGALOP
The parameter **idxDesc** or **idxInfo** is incorrect.

o2x_spatial_index_free_descriptor

Summary Free the memory used by a spatial index descriptor.

Syntax

```
#include <o2_error.h>
#include <o2_spatialindex.h>
int o2x_spatial_index_free_descriptor(
                            o2x_SpatialIndexDesc** idxDesc);
```

Description Free the memory associated with the spatial index descriptor created by **o2x_spatial_indexed**. Every spatial index descriptor should be freed by **o2x_spatial_index_free_descriptor.**

After a call to **o2x_spatial_index_free_descriptor** a spatial descriptor is always NULL.

Returns **O2_OK** if successful, error code otherwise.

Errors **O2E_PARAMETER**
The spatial index descriptor **idxDesc** is incorrect.

o2x_spatial_index_get_collection

Summary Return the handle of the indexed collection.

Syntax

```
#include <o2.h>
#include <o2_error.h>
#include <o2_spatialindex.h>
Handle o2x_spatial_index_get_collection(
                          o2x_SpatialIndexDesc* idxDesc);
```

Description Return the handle of the collection of persistent objects associated with the spatial index identified by **idxDesc.**

Returns A valid collection handle if successful, a NULL handle otherwise.

o2x_spatial_index_insert_key

Summary Insert a new entry in the spatial index.

Syntax

```
#include <o2.h>
#include <o2_error.h>
#include <o2_spatialindex.h>
int o2x_spatial_index_insert_key(
                    o2x_SpatialIndexDesc *idxDesc,
                    int keyLength, void *key,
                    Handle element);
```

Description Inserts an entry (**key**, **element**) into the spatial index.

The spatial key is passed by the parameter **key** which provides the value and the parameter **keyLength** which provides the length. The type of key is not defined at the level of the O_2Engine and O_2Store. The spatial indexing module provides the structure of the key when needed at the level of the application.

When using the standard O_2Spatial indexing engine, the key is expected to be a pointer to a **SiBoundingBox** object, and **keyLength** should be equal to zero.

The parameter **element** is a handle of the O_2 object to be indexed.

This function fails if a **scan** operation is active on the spatial index.

Returns **O2_OK** if successful, error code otherwise.

Errors

O2E_NOTSUPPORTED
The function is not implemented.

O2E_SIMALGO
Error in Spatial Index Module algorithm.

O2E_ILLEGALCALL
Function was called during a scan.

O2E_PARAMETER
The parameter **idxDesc** or **key** or **element** is incorrect.

O2E_KEYALREADYEXISTS
The couple (**key**, **element**) already exists in the index.

O2E_NOSPACEONDISK
No more space on disk.

See Also

`o2x_spatial_index_group_insert_key()`
`o2x_spatial_index_delete_key()`
`o2x_spatial_index_group_delete_key()`
`o2x_spatial_index_replace_key()`

o2x_spatial_index_group_insert_key

Summary Insert multiple new entries in the spatial index.

Syntax

```
#include <o2.h>
#include <o2_error.h>
#include <o2_spatialindex.h>
int o2x_spatial_index_group_insert_key(
                    o2x_SpatialIndexDesc *idxDesc,
                    int card, int keyLength,
                    void **keyArray, Handle *eltArray);
```

Description Inserts multiple entries (**key[i]**, **eltArray[i]** with **i** varying between [**0**, **card-1**]) into the spatial index, with minimization of the traversal of the spatial index tree.

The parameter **keyLength** gives the length of a single key. There is no definition of the key type in the O_2Engine and O_2Store. The spatial indexing module provides the structure of the key for the application when needed.

When using the standard O_2Spatial indexing engine, **keyArray** is expected to be an array of pointers to **SiBoundingBox** objects, and **keyLength** should be equal to zero.

This function fails if a **scan** operation is active on the spatial index.

Returns **O2_OK** if successful, error code otherwise.

Errors

O2E_NOTSUPPORTED
The function is not implemented.

O2E_SIMALGO
Error in Spatial Index Module algorithm.

O2E_ILLEGALCALL
Function was called during a scan.

O2E_PARAMETER
The parameter **idxDesc** or **keyArray** or **eltArray** is incorrect.

O2E_NOSPACEONDISK
No more space on disk.

See Also **o2x_spatial_index_insert_key()**
o2x_spatial_index_delete_key()
o2x_spatial_index_group_delete_key()
o2x_spatial_index_replace_key()

o2x_spatial_index_delete_key

Summary Remove an entry from the spatial index.

Syntax

```
#include <o2.h>
#include <o2_error.h>
#include <o2_spatialindex.h>
int o2x_spatial_index_delete_key(
                    o2x_SpatialIndexDesc *idxDesc,
                    int keyLength, void *key,
                    Handle *element);
```

Description Removes the couple (**key, element**) from the spatial index.

The spatial key is passed by the parameter **key** which provides the value and the parameter **keyLength** which provides the length. The type of key is not defined at the level of the O_2Engine and O_2Store. The spatial indexing module provides the structure of the key when needed at an application level.

When using the standard O_2Spatial indexing engine, the key is expected to be a pointer to a **SiBoundingBox** object, and **keyLength** should be equal to zero.

The parameter **element** is a handle for the O_2 indexed object.

This function fails if a **scan** operation is active on the spatial index.

Returns **O2_OK** if successful, error code otherwise.

Errors

O2E_NOTSUPPORTED
The function is not implemented.

O2E_SIMALGO
Error in Spatial Index Module algorithm.

O2E_ILLEGALCALL
Function was called during a scan.

O2E_PARAMETER
The parameter **idxDesc** or **key** or **element** is incorrect.

O2E_KEYNOTFOUND
The couple (**key**, **element**) already exists in the index.

See Also

o2x_spatial_index_insert_key()
o2x_spatial_index_group_insert_key()
o2x_spatial_index_delete_key()
o2x_spatial_index_replace_key()

o2x_spatial_index_group_delete_key

Summary Remove multiple entries from the spatial index.

Syntax

```
#include <o2.h>
#include <o2_error.h>
#include <o2_spatialindex.h>
int o2x_spatial_index_goupr_delete_key(
                    o2x_SpatialIndexDesc *idxDesc,
                    int card, int keyLength,
                    void **key, Handle *eltArray);
```

Description Removes multiple entries (**key[i]**, **eltArray[i]** with **i** varying between [**0**, **card-1**]) from the spatial index, with minimization of the traversal of the spatial index tree.

The parameter **keyLength** gives the length of a single key. There is no definition of the key type in the O_2Engine and O_2Store. The spatial indexing module provides the structure of the key for the application when needed.

When using the standard O_2Spatial indexing engine, **keyArray** is expected to be an array of pointers to **SiBoundingBox** objects, and **keyLength** should be equal to zero.

This function fails if a **scan** operation is active on the spatial index.

Returns **O2_OK** if successful, error code otherwise.

Errors

O2E_NOTSUPPORTED
The function is not implemented.

O2E_SIMALGO
Error in Spatial Index Module algorithm.

O2E_ILLEGALCALL
Function was called during a scan.

O2E_PARAMETER
The parameter **idxDesc** or **keyArray** or **eltArray** is incorrect

O2E_NOSPACEONDISK
No more space on disk.

See Also **o2x_spatial_index_insert_key()**
o2x_spatial_index_group_insert_key()
o2x_spatial_index_delete_key()
o2x_spatial_index_replace_key()

o2x_spatial_index_replace_key

Summary Substitute one entry with another entry in the spatial index.

Syntax

```
#include <o2.h>
#include <o2_error.h>
#include <o2_spatialindex.h>
int o2x_spatial_index_replace_key(
                    o2x_SpatialIndexDesc *idxDesc,
                    int keyLength, void *key,
                    Handle oldEltHd, Handle newEltHd);
```

Description Substitutes the couple (**key**, **oldEltHd**) by the couple (**key**, **newEltHd**) in the spatial index.

The spatial key is passed by the parameter **key** which provides the value and the parameter **keyLength** which provides the length. The type of key is not defined at the level of the O_2Engine and O_2Store. The spatial indexing module provides the structure of the key when needed at an application level.

When using the standard O_2Spatial indexing engine, the key is expected to be a pointer to a **SiBoundingBox** object, and **keyLength** should be equal to zero.

The parameters **oldEltHd** and **newEltHd** are handles for O_2 objects.

This function can execute if a **scan** operation is active on the spatial index.

Returns **O2_OK** if successful, error code otherwise.

Errors

O2E_NOTSUPPORTED
The function is not implemented.

O2E_SIMALGO
Error in Spatial Index Module algorithm.

O2E_PARAMETER
The parameter **idxDesc** or **key** or **oldEltHd** or **newEltHd** is incorrect.

O2E_KEYNOTFOUND
The couple (**key**, **oldEltHd**) does not exist in the index.

O2E_KEYALREADYEXISTS
The couple (**key**, **newEltHd**) already exists in the index.

O2E_NOSPACEONDISK
No more space on disk.

o2x_spatial_index_scan_open

Summary Start a search operation on the spatial index.

Syntax

```
#include <o2_error.h>
#include <o2_spatialindex.h>
int o2x_spatial_index_scan_open(
                    o2x_SpatialIndexDesc *idxDesc,
                    int operation, int areaLength,
                    void *areaValue,
                    o2x_SpatialIndexScan **idxScan);
```

Description Initializes a scan on the spatial index identified by **idxDesc** and returns an external identifier for it as the variable **idxScan**.

The parameters **operation**, **areaLength** and **areaValue** define the spatial predicate which is applied to filter the entries returned by the function **o2x_spatial_index_scan_read**.

The available spatial operations are defined by a code at the application level. This code is transferred by the parameter **operation**.

When using the O_2Spatial indexing engine, the possible values for the parameters **operation** and **areaValue** are as follows :

Operation parameter	Meaning	areaValue parameter
O2X_CONTAIN	Retrieve all the objects which contain a given point.	A pointer to a **SiBoundingBox** object which contains the requested point.
O2X_INTERSECT	Retrieve all the objects which intersect a given rectangle	A pointer to a **SiBoundingBox** object which contains the requested rectangle
O2X_IS_CONTAINED	Retrieve all the objects which are inside a given rectangle	
O2X_ELLIPSE	Retrieve all the objects which intersect a given ellipse	A pointer to a **SiEllipse** object which contains the requested ellipse.
O2X_ELLIPSE_STRICT	Retrieve all the objects which are inside a given ellipse	

Operation parameter	Meaning	areaValue parameter
O2X_POLYGON	Retrieve all the objects which intersect a given polygon.	A pointer to a **SiPolygon** object which contains the list of the requested polygon vertices.
O2X_POLYGON_STRICT	Retrieve all the objects, which are inside a given polygon	
O2X_POLYLINE	Retrieve all the objects which intersect a given polyline	A pointer to a **SiPolyline** object which contains the list of the requested polyline vertices.

Returns

O2_OK
if successful, error code otherwise.

Errors

O2E_SIMALGO
Error in Spatial Index Module algorithm.

O2E_PARAMETER
The parameter **idxDesc**, **operation** or **areaValue** is incorrect.

Samples

```
SiBoundingBox requestedPoint(x, y, x+1, y+1);
if (o2x_spatial_index_scan_open (
       idxDesc,
       (int) O2X_CONTAIN,
       sizeof (SiBoundingBox *),
       &requestedPoint,
       &idxScan)  !=O2_OK)  {
    return-1;       //Error;
}
```

```
Si3DPoint center(x, y);
SiEllipse requestedEllipse(center, xRadius, yRadius);
if (o2x_spatial_index_scan_open (
            (idxDesc,
            (int) O2X_ELLIPSE,
            sizeof  (SiEllipse*)
            &requestedEllipse,
            &idxScan)    !=O2_OK)
    return -1;
}
```

```
Si3DPoint* pointArray=...;
SiPolyline requestedPolyline(numPoints, pointArray);
if (o2x_spatial_index_scan_open  (
            idxDesc,
            (int) O2X_POLYLINE,
            sizeof (SiPolyline*)
            &requestedPolyline,
            &idxScan)  !=O2_OK)  {
    return -1;
}
```

See Also `o2x_spatial_index_scan_close()`
`o2x_spatial_index_scan_read()`

o2x_spatial_index_scan_close

Summary Close a search operation on the spatial index.

Syntax

```
#include <o2_error.h>
#include <o2_spatialindex.h>
int o2x_spatial_index_scan_close(
                        o2x_SpatialIndexScan **idxScan);
```

Description Liberates the cursor of the scan **idxScan**.

Returns **O2_OK** if successful, error code otherwise.

Errors

O2E_SIMALGO
Error in Spatial Index Module algorithm.

O2E_PARAMETER
The parameter **idxScan** is incorrect.

O2E_BADSCANID
The parameter **idxScan** does not exist.

See Also **o2x_spatial_index_scan_open()**
o2x_spatial_index_scan_read()

o2x_spatial_index_scan_read

Summary

Return the next element of the spatial index which satisfies the operation of the current spatial scan.

Syntax

```
#include <o2_error.h>
#include <o2_spatialindex.h>
int o2x_spatial_index_scan_read(
                    o2x_SpatialIndexScan *idxScan,
                    o2x_ScanDirection direction,
                    Handle *hd);
```

Description

Moves the scan cursor forward if the parameter **direction** is set to **O2X_NEXT** or backward if **direction** is set to **O2X_PREVIOUS**, and returns in the parameter **hd** a handle to an indexed object pointed to by the scan cursor.

Any other value for the parameter **direction** is incorrect.

Returns

O2_OK if successful, otherwise an error code; notably **O2E_KEYNOTFOUND** which signifies that there are no more entries to recover.

Errors

O2E_NOTSUPPORTED
The function is not implemented.

O2E_SIMALGO
Error in Spatial Index Module algorithm.

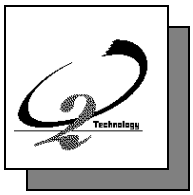
O2E_PARAMETER
The parameter **idxScan** or **direction** or **hd** is incorrect

O2E_BADSCANID
The parameter **idxScan** does not exist.

O2E_KEYNOTFOUND
No more accessible entries.

See Also

o2x_spatial_index_scan_open()
o2x_spatial_index_scan_close()



INDEX

A

B

C

D

F

G

I

J

L

O

R

S