# Tutorial: Package Index Structures

## Introduction:

In this tutorial we give a brief overview about the index structures package in XXL-Java library. We start our discussion with a presentation of the core architecture of the tree-based index structures. Then we go through basic initialization steps and discuss how to set up queries on indexes. By the end of the first section we discuss advanced topics about the bulk-loading index structures. In the last section we give a step-by-step description how to initialize and use the framework. We show how to initialize B+trees, R-trees and M-trees.

## Basic Concepts of Index Structures in XXL-Java Library:

XXL-Library comes along with a reach and powerful demand-driven cursor framework for data processing and various implementations of index structures. The package xxl.core.indexStructures contains a rich collection of tree-based index structures like various kinds of B+trees, R-trees and M-trees. Instead of implemeneting each of these structures one by one, we decided to provide a framework that allows an easy implementation of new structures.

The framework is based on the concept of grow-and-post trees ( TODO cite). All index structure classes are inherited from basic class xxl.indexStrutures.Tree.

The abstract class tree defines two core internal classes: Index Entry (Tree.IndexEntry) and Node (Tree.Node). The one of the most important components is the Interface xxl.indexstructures.Descriptor. The objects of this class are used as “keys” for navigation, traversing and querying. E.g R-tree uses as Descriptor MBR, M-tree a sphere and B+tree one dimensional comparable objects. In other words, a descriptor is a representation value of the object to store. The actual index is built using the “extracted” keys.

Interface Descriptor provides several methods which are need during the traversing the index structure. They are managed by IndexEntry Objects. Consider following case we want to store a POJO (Plain Old Java Object) Person: which have two fields PersonalNumber of Type Integer and Name of Type String and we want to create an B+tree Index on personal number then the Descriptor is a Integer Values which corresponds to a personal number.

Another example if we want to manage 2D points in XXL: assume we deal with a DoublePoint Object “P” which internally holds an array of double, then the descriptor is a DoublePointRectangle (P,P). All index structure become as parameter an object reference to a Function object which implements Interface xxl.core.functions.Function (see Code Example 1). This tells the index structure how to extract (or to map) a “key” (descriptor) of this object.

**public** Function<DoublePoint, DoublePointRectangle> *getDescriptor* =

**new** AbstractFunction<DoublePoint, DoublePointRectangle>() {

**public** DoublePointRectangle invoke(DoublePoint p) {

**return** **new** DoublePointRectangle(p, p);

}

};

Code Example 1

All implemented index structures in package are leaf search trees. It means that index nodes manage index entries which themselves stores a pointer (id) to the node and object descriptor. The leaf nodes can manage arbitrary objects. We distinguish between leaf nodes and index nodes based on Level value. In our implementation the leaf nodes are Level = 0 nodes, otherwise they are index nodes.

The nodes of index structures are stored in container, which is another abstraction layer in our library. Container is a generic interface for storing arbitrary objects. It supports basic operation *Insert, Remove* and *Update*. The *insert* method of the containers returns an ID-Object; this can be used to by the *get* method. Actually Container Interface can be seen as a simple Key-Value storage.

For using the index structure for external memory the BlockFileConatiner can be used. This container manages blocks or pages of data with a fixed sized byte size. Block in our library is implemented as continues array of bytes. In order to convert an arbitrary java object in a byte sequence interface *converter* should be used. This interface extends native java serializable interface.

All containers can be plugged in each other (actually they implement decorator patter) building a chain of containers. The highest container in this chain should be ConverterContainer which is parameterized with a Converter; it capsules methods about mapping (serialiazing) to byte stream (e.g. Nodes to Blocks). Additionally other containers can be used common scenario is to use BufferedContainer which is parameterized with a generic buffer e.g. LRUBuffer. The implementation is very generic so it can be plugged in chain of containers on different places e.g. we can put it before ConverterContainer or after depending on the place where buffer resides it manages different type of objects e.g. Nodes or Block resp. (In order to count I/O – Operations CounterConatainer can be used it manages internal counter about get, insert and update operations issued on containers see Figure 1). The code example Code Example 2 shows common initialization pattern for containers.



Figure 1

For mapping the tree nodes to blocks, all index structures provide NodeConverter object. This class is parameterized with a converter for the data and descriptor converter. So for initialization of the index structure two converters should be provided one for object which should be managed by a leaf nodes and one for “keys” which are managed in index entries in in internal nodes.

Container blockConatainer = **new** BlockFileContainer("D:/tree/rtree", 4096);

Container IOCounterContainer = **new** CounterContainer(blockConatainer);

Container bufferContainer = **new** BufferedContainer(IOCounterContainer, **new** LRUBuffer(50));

Container treeConatainer = **new** ConverterContainer(bufferContainer,

tree.nodeConverter(objectConverter, indexEntryConverter));

Code Example 2

Most of index structures in XXL library require that size of serialized object is bounded by the certain value the same goes for descriptor objects. The reason is a fixed size of blocks. Before we initialize the particular index structure we need to provide the max size of serialized data objects and descriptors. Based on these values the structural reorganizations are triggered (split, merge). In particular two predicates will be instantiated NodeOverflows and NodeUnderflows.

In following we will use notion of block and node as synonyms. The common layout of the block (node) is a header and continues array of entries. In case of leaf node they are data objects. Internal nodes manage IndexEntry in this array. In most cases the header of the node consists of following values:

1. Level information (needs 2 byte, stored as a short java type). If level = 0 codes the leaf node, otherwise it is index node.
2. Information about the number of entries (needs 4 byte, stored as integer value)

In some cases there is also additional information stored e.g. neighbor linkage in case of B+tree. Thus if we have a block of size 4096 bytes (4KB) at least 6 bytes is a header information. Remain part is used for storing the entries array. Index entries bear additional information which will be also serialized. The common layout of the indexEntry is:

1. Id of a block to which the index entry points (in most cases it is a java long value which has 8 byte serialized size)
2. Descriptor of data object stored in leaf nodes

It means that the serialized size of index entry is a sum of id + descriptor size.



Consider example above where we wanted to store 2D double points in leafs of say R-tree and we use as descriptor doublePointRectangles. The 2D doublePoint object has a fixed size of dimension\*8 bytes pro double value. Descriptor DoublepointRectangle stores two points lower left corner and the higher right corner this result in 2\*(size of double point). Thus the maximum number entries pro leaf block is (block Size – header size )/object size. We can store 4090/16 doublePoint pro leaf block. The miximal number of indexEntries are then (block Size – header size ) divided by the size of index entry (id + descriptor size). Then we can manage 4090/(8+16\*2) index entries pro index block.

Let us summarize the information need for a first initialization of index structure:

**Basic Initialization Steps:**

1. Write a converter for data
2. Write a converter for a Descriptor (e.g. for MBR doublePointRectangle)
3. Write a Map function, which extract a key (descriptor) from an object
4. Initialize Containers for index structure
5. Define max sizes in bytes for data and converter

Let us take a closer look at a common initialization method defined in Tree class. Class Tree offers several init methods, as example we consider only one of them. Advanced init routines will be discussed in following sections:

***public*** *Tree initialize (Function getDescriptor,* ***int*** *blockSize, Container container,* ***int*** *dataSize,* ***int*** *descriptorSize,* ***double*** *minMaxFactor)*

First argument is reference to a function which extracts a “key” from an object to store. The second argument is a block size this is needed to compute the maximal and minimal capacity of leaf nodes and index nodes. Then we pass a container reference in which are the nodes of tree will be stored. Additionally we need a maximal data size in bytes, maximal descriptor size in bytes and factor which defines a minimal capacity of the nodes e.g. 0.5 for a B+Tree or 0.33 for R-tree.

In the next paragraph we discuss das reloading and re-initialization of index structure. Briefly the reloading the recent state of index structure is done upon saving the information about the root node. In following we summarize the steps need to store a current state of index structure:

1. Extract Information about IndexEntry which represent the root entry of the tree. Class tree provides method tree.rootEntry().
2. Extract
3. store level (height)
4. store address (id) of the root node
5. store a descriptor of the root node as long with a descriptor of index structure (in most cases they are equal but e.g. BTree index structure descriptor is an interval)

### Querying:

Exact match query; Range query; Nearest Neighbor query; Skyline query;

### Bulk-Loading:

Sort-based bulk loading; External Sorter;

## Examples:

### BPlusTree:

#### First Initialization and Querying:

#### Reload:

#### Bulk-Loading:

### RTree:

#### First Initialization and Querying:

#### Reload:

#### Bulk-Loading:

### MTree:

#### First Initialization and Querying:

#### Reload:

#### Bulk-Loading: