Department of Informatics, University of Zürich

BSc Thesis

Implementing a disk-resident spatial index structure (Quadtree)

Freddy Panakkal

Place of birth: Schaffhausen, Switzerland

Matrikelnummer: 12-737-813

Email: freddy.panakkal@uzh.ch

June 10, 2016

supervised by Prof. Dr. M. Böhlen and G. Garmpis

University of Zurich[UZH]

Department of Informatics

Acknowledgements

Most of all, I would like to thank my supervisor Georgios Garmpis for his helpful support, and for the feedback and guidance he provided. Also, I would like to thank Prof. Dr. Michael Böhlen for giving me the opportunity to write my bachelor thesis at the Database Technology Group of the University of Zurich.

Abstract

In the last decades more and more systems are dealing with multi-dimensional data. An example for such a system are the geographic information systems (GIS), like Google Maps or PostGIS, that store, manipulate and analyze geographic data. It is the task of the underlying database to deal with such huge amount of geographic information. One way to deal with multi-dimensional data is by using a tree data structure like the quadtree. A quadtree works similar to a binary tree but it is designed for two-dimensional data. This thesis discusses the implementation of a disk-resident quadtree and analyzes the efficiency of a quadtree. For this purpose, a buffer manager has been implemented and a quadtree on top of it. By performing different experiments, good and bad scenarios of using a quadtree and a buffer manager are demonstrated.

Zusammenfassung

In den letzten Jahrzehnten entstanden immer mehr Systeme, welche mit mehrdimensionalen Daten arbeiten. Geoinformationssysteme (GIS), wie Google Maps oder PostGIS, sind Beispiele für solche Systeme, mit welchen geographische Daten erfasst, bearbeitet und analysiert werden. Eine Anforderung an solche Systeme ist die effiziente Handhabung riesiger Mengen mehrdimensionaler Daten. Eine Möglichkeit mehrdimensionaler Daten strukturiert abzulegen ist mittels einer Baumstruktur, wie einem Quadtree. Quadtrees funktionieren ähnlich wie Binärbäume, jedoch mit zweidimensionalen Schlüsseln. Diese Arbeit befasst sich mit der plattenresidenten Implementierung eines Quadtrees und der Analyse der Performance. Zu diesem Zweck wurde zuerst ein Buffer Manager implementiert und aufbauend auf diesem der Quadtree. Durch verschiedene Experimente werden sowohl gute als auch schlechte Szenarien dieser Implementierung aufgezeigt.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Nowadays, there are many applications which are using spatial data. Systems that are designed to store, manipulate, analyze etc. geographic data are called geographic information systems (GIS). Google Maps or PostGIS are examples for GIS. There are also many custom-designed GIS for firms for their electrical grids, rails, installed systems etc. The performance of such applications is crucial for end users.
There are different data structures which are designed to handle big amounts of spatial data. One of them is the quadtree. A quadtree works similar to a binary tree. But instead of using a one-dimensional key, the quadtree is designed for two-dimensional keys like coordinates. Because of the nature of quadtree, they are predestined for range searches [FB74]. Finkel and Bentley [FB74] invented quadtree and performed some experiments in insertions and range searches for small numbers of points. Experiments for more points might be very interesting, since building tree structures makes sense for big data.
At present there are not many in-memory databases. Most databases store their data in disk and load the needed part into memory. For this a buffer manager is used, which loads only a part of disk resident data to the buffer in memory. With a predefined strategy the buffer manager tries to minimize the I/O operations on the disk.
A system which handles multi-dimensional data usually has on the one hand large data sets and on the other hand limited main memory. Therefore, it would be interesting to implement a quadtree on top of a buffer manager and observe the impacts of this combination.

1.2 Problem statement

The focus of this thesis is to show advantages and disadvantages of a quadtree implemented on top of a buffer manager. For this purpose, first a buffer manager and second a quadtree are implemented. By using this implementation of a disk-resident quadtree for several experiments on insertions and searches, good and bad scenarios are demonstrated. These scenarios are detected by varying different parameters of both the buffer manager and the quadtree.

1.3 Thesis Outline

This thesis is structured as follows. In Chapter 2 the related work is presented. The nature of a quadtree according to Finkel and Bentley [FB74] and the function of a buffer manager is explained. In the following Chapter 3 the implementation of the buffer manager and the

quadtree are discussed. Chapter 4 describes the design of the experiments. In Chapter 5 the results of the experiments are presented and discussed. Finally, in Chapter 6, the thesis is summarized and reflected.

2 Related Work

2.1 Quadtree

In this section, the definition of a quadtree according to Finkel and Bentley [FB74] is described. The quadtree described by Finkel and Bentley [FB74] and used in this thesis is also called point quadtree [Sam84] since it is focused on storing only points and not other more complicated data types like lines and regions. There are also different modifications of quadtrees so for example the region quadtree [SW84] or the edge quadtree [Shn81], but this modifications are not discussed in this thesis.

2.1.1 Definition

Quadtree is a tree data structure for storing data with two-dimensional keys. It is an adaptation of binary tree. Coordinates are an example for such two-dimensional keys. In point quadtree every node stores exactly one record and has four child nodes. A node divides the space into four quadrants. These quadrants are bordered by a vertical and a horizontal axis specified by the point of the node. Each child node corresponds to one of the four quadrants and is referred with the intercardinal directions NE, NW, SW, SE.
The first inserted node is stored as the root of the quadtree and divides the plane into four quadrants. The next inserted node is assigned to one of the four child nodes of the root. According to the relative position of the new node to the root, it is assigned to the matching child node of the root. For example, in case the matching child node is already occupied by another node, the whole procedure is repeated recursively for this matching child node instead of the root until an unoccupied child node is found. The algorithm for insertion is explained in more detail in the next section.

2.1.2 Insertion

The first node inserted into a quadtree is the root. Algorithm 1 describes the way of inserting a new node `K` into a quadtree with root `R` [FB74]. First, the function `compare` is used to specify in which of the four quadrants `K` lies and the result is stored in the integer called `direction`. `direction` equals 1 means NE, 2 means NW, 3 means SW and 4 means SE. Next it is checked if there is already a node assigned at the child specified by `direction`. If yes, the procedure is repeated with the subtree. For this, the corresponding child node is handled as root `R` and the new direction of `K` to `R` is checked. This is done until a `R` with an unassigned child node at position `direction` is found. As soon as this has been found, `K` is

assigned as the corresponding child of R.

Data: NODE K; ROOT R
Result: Node K inserted in quadtree with root R
INTEGER DIRECTION; COMMENT: Direction from parent to child, i.e.1,2,3, or 4;
$DIRECTION \leftarrow COMPARE(R; K)$;
while *R[DIRECTION]* $\neq$ *NULL* **do**
  COMMENT: Each iteration dives one level deeper;
  $R \leftarrow R[DIRECTION]$;
  $DIRECTION \leftarrow COMPARE(R; K)$;
  if *DIRECTION = 0* **then**
    RETURN; COMMENT: Node already exists;
  end
end
$R[DIRECTION] \leftarrow K$;

Algorithm 1: Insertion

Figure 2.1 illustrates a quadtree generated by inserting seven points in the following order: A,B,C,D,E,F,G. On the right side of the figure the tree structure is shown with the root (A) and all the child nodes. How the space is divided into quadrants by the seven points is illustrated on the left side of the figure.

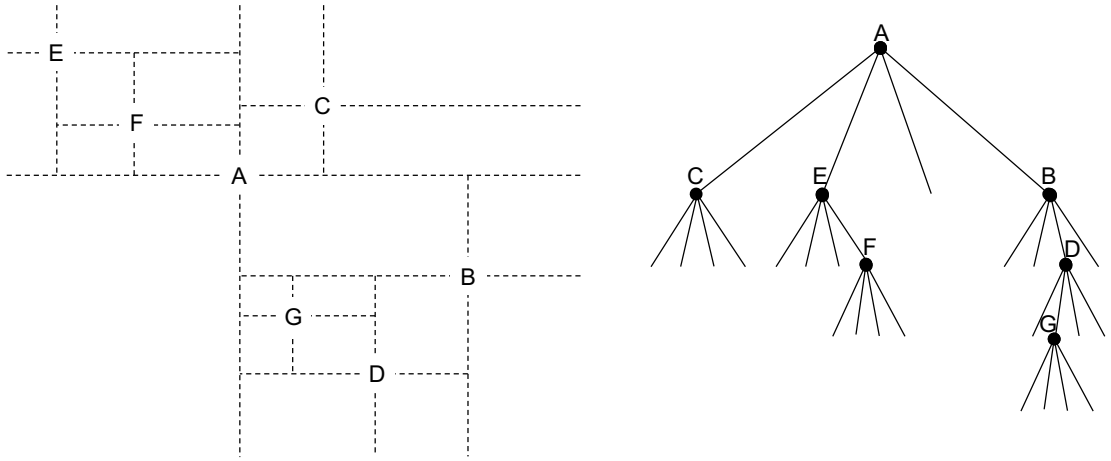


Figure 2.1: Example of a quadtree; the mapping and the tree structure

2.1.3 Balancing

Using the aforementioned insertion algorithm by Finkel and Bentley [FB74] can result in an unbalanced quadtree. The more unbalanced a tree gets, the deeper will be the structure of this tree. The depth of a node is defined as the number of ancestors of this node and therefore the number of edges from the node to the root [GTM07]. The biggest depth is also referred as the height of the tree. A tree is balanced if it has the minimum possible height. Therefore, the height of a balanced tree can not be lowered by rearranging its nodes. A balanced tree has the

advantage that on average less nodes have to be visited in order to find a node. So insertions and searches can be done more efficiently. Figure 2.2 shows two quadtrees with the exact same points; on the left an unbalanced tree and on the right a balanced one.

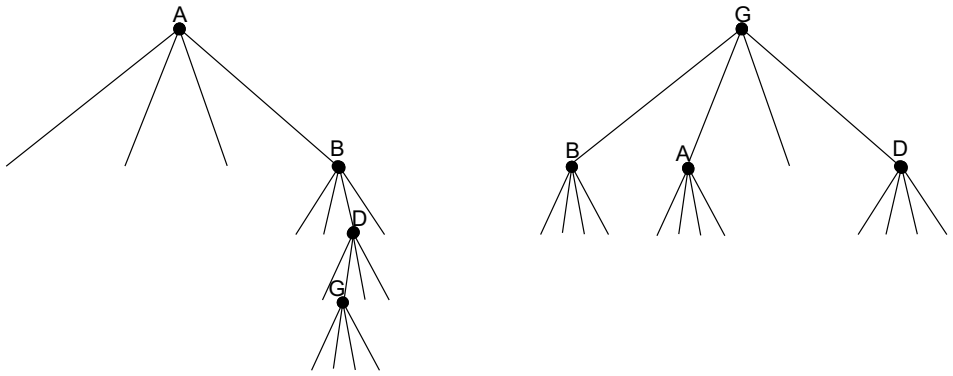


Figure 2.2: Unbalanced and balanced quadtree

To avoid unbalanced trees Finkel and Bentley [FB74] described a method to balance the newly created leaf after an insertion. This does not balance the full tree but only the new leaf. As it only balances leaves this method does not result in a fully balanced tree but reduces the average depth of nodes. This simple balancing algorithm described by Finkel and Bentley [FB74] can only be applied if the parent of the new node and the parent of the parent have exactly one child node assigned. In this case the three nodes can be rearranged so that the subtree is more balanced. Two types of balancing are defined; single balance and double balance. Assume three nodes `A`, `B`, `D`, where `D` is child of `B` and `B` child of `A`. Double balance is when `D` lies within the rectangle set by corners in `A` and `B`. In this case the node `D` is rearranged to the position of `A` and `A` and `B` are set as `D`'s children. Figure 2.3 illustrates this double balance. Single balance is if `D` does not lie within the rectangle bounded by `A` and `B`. In this case `B` takes the position of `A` and `A` and `C` are assigned as its children. This single balance is illustrated in Figure 2.4.

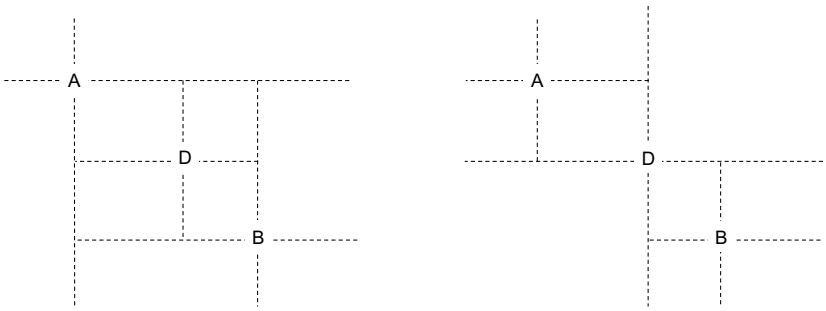


Figure 2.3: Double balance: unbalanced and balanced leaf

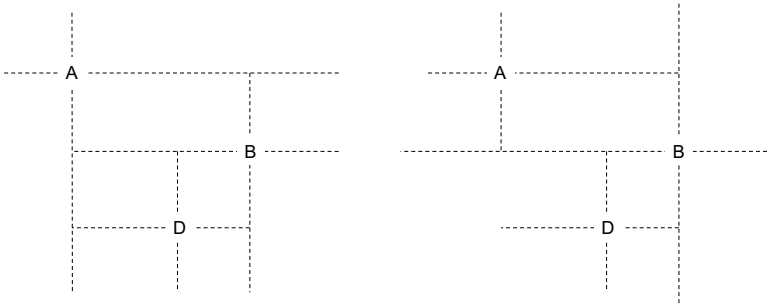


Figure 2.4: Single balance: unbalanced and balanced leaf

2.1.4 Region Search

Finkel and Bentley [FB74] described an algorithm to get all nodes which lay in a specified rectangular region. Starting from the root only those child nodes, whose regions overlap the search rectangle are processed. So in a recursive way, all nodes are found without processing every single node.
Algorithm 2 illustrates the way how all nodes within a region can be found. The input for a search is the root `P` of the quadtree and the region to be searched, bounded by the four values `L`(left), `R`(right), `B`(bottom), `T`(top). At first, the root is checked by taking the x and y coordinates of `P`. If x is between `L` and `R` and y between `B` and `T`, `P` lies in the search region and is returned. Next all four quadrants are checked to determine if their region overlap the search rectangle. If a quadrant does not overlap the search rectangle, the corresponding subtree contains no relevant nodes for the search. But if a quadrant overlaps the rectangle the corresponding subtree is searched recursively. For this a new region search with the corresponding child node as root is started. The rectangle of this search is the intersection of the prior search rectangle and the quadrant of the child node. So, the new region is build by the x and y coordinates of the prior root and depending the quadrant either the `L` or `R` and either the `B` and `T` of the prior search region.

Data: ROOT P; REAL L, R, B, T
Result: Find all nodes of quadtree with root P within the window bounded by L (left), R (right), B (bottom), T (top)
REAL XC, YC; COMMENT: X and Y coordinates of P;
$XC \leftarrow X(P)$;
$YC \leftarrow Y(P)$;
if *InRegion(XC, YC)* **then**
  Found(P);
end
if *P[1]* $\neq$ *NULL and RectangeOverlapsRegion(XC, R, YC, T)* **then**
  RegionSearch(P[1], XC, R, YC, T)
end
if *P[2]* $\neq$ *NULL and RectangeOverlapsRegion(L, XC, YC, T)* **then**
  RegionSearch(P[2], L, XC, YC, T)
end
if *P[3]* $\neq$ *NULL and RectangeOverlapsRegion(L, XC, B, YC)* **then**
  RegionSearch(P[3], L, XC, B, YC)
end
if *P[4]* $\neq$ *NULL and RectangeOverlapsRegion(XC, R, B, YC)* **then**
  RegionSearch(P[4], XC, R, B, YC)
end

Algorithm 2: Region Search

2.1.5 Evaluation

Finkel and Bentley [FB74] tested both the insertion and region search and collected various quantitative data. The insertion was tested with uniformly distributed random points and with tree sizes varying from 25 nodes until 10'000 nodes. The total path length (sum of depth of all nodes) and its standard deviation of the resulting trees were reported for these experiments. This was done for the simple insertion and the insertion with leaf-balancing. The experiments showed that the average path length of a quadtree with N points is roughly proportional to log N. They conclude that for searching one point in the quadtree can be expected to take log N probes. These experiments also showed that the insertion with balancing led to lower average path length than the insertion without balancing.
For the experiments with the region search, they randomly generated quadtrees for six tree sizes ranging from 125 nodes to 4000 nodes. These quadtrees were used for the region searches. Randomly generated rectangles with in total six different edge sizes were used as search regions. For every region search the number of visited nodes and the number of found were documented. Finkel and Bentley [FB74] conclude that the insertion algorithm yields N log N performance for random keys and that region searching is quite efficient. They also state that the concept of a quadtree can easily be adapted to any number of dimensions.

2.2 Paging

The records of a database must be stored physically on some storage medium [EN10]. There are different types of storage with different advantages and disadvantages. Storage media form a storage hierarchy depending their access time and capacity, which includes two main categories. Primary storage is very fast and the CPU can operate on it directly. But the capacity is very limited and it is very costly. Main memory is an example of a primary storage. The other main category is secondary storage like disk storage. It is slower than primary storage but has a larger capacity and is cheaper.
A database management system deals with storage media on different levels. Data is usually too large to fit entirely in main memory [EN10]. Paging is used to retrieve and modify data in main memory but store it on disk space. Data on disk is partitioned into fixed-length blocks. When the user or system wants to access a block the first time, the corresponding block is copied from the disk to a page in memory. A page in memory holds the space for one full block. Normally there is a fixed number of pages. All these pages together can also be seen as the buffer. So if a block is requested, first it is checked, if the block is in the buffer. If yes, the page in memory is accessed and there is no need to make a read in the disk. If it is not found in the buffer, then the block in the disk is copied to the buffer, which is also called effective read. If a write of a block is requested, the writing is done in a page in memory. It is not written to the disk immediately. The moment this page has to be freed (i.e. before shut down or to be replaced by another block) it is written to the disk, this is called effective write.
If all pages are filled and a new block from the disk is requested, the system empties the most dispensable page. There are several techniques for this page replacement, like first-in/first-out, least recently used or not frequently used. The least recently used (LRU) page replacement algorithm, which is also used for the implementation, replaces the page which was the least used by the system. According to the principle of locality (related storage locations are frequently accessed) effective reads and writes are expected to be much less than the requested ones. So using such a memory buffer disk I/O operations are reduced significantly.

3 Implementation

In order to use and test the quadtree index structure also for a big number of nodes, first a buffer manager has been implemented. This buffer manager is implemented in a way that it could also be used for other purposes than for a quadtree. In a second step, the quadtree logic has been implemented on top of the buffer manager. All the coding is done in C++.

3.1 Buffer Manager

The buffer manager is designed to create and handle files that do not entirely fit into memory. These files are partitioned into blocks of fixed size. The maximum number of blocks that can be loaded in memory at the same time is specified at the initialization of the buffer manager. The buffer manager provides functions to (i) create, (ii) open, (iii) close and (iv) delete files and also to (v) add, (vi) read, (vii) write and (viii) delete blocks. For this four classes are implemented. Figure 3.1 shows the UML class diagram of the buffer manager.
The buffer manager uses the LRU procedure as replacement policy. The LRU replacement policy can be briefly descried as follows. There exists a global LRU counter initialized to 1 and every time a block is accessed (read or written) the value of the global LRU counter is assigned to this page and the global LRU counter is increased. If a page is requested and all pages are already used, the page with the lowest LRU value is replaced.

3.1.1 The file structure

A file created by the buffer manager is partitioned into blocks. It begins with a header block, which does not store content of the file. The header block stores the number of blocks the file has (`numberOfBlocks`) and a bitmap to know which blocks are valid (`validBlocks`). This bitmap is provided by the library `<bitset>`. The size of this header block depends on the global constant `MAXBLOCKS`, which determines the maximum number of blocks a file can contain so it also specifies the size of the bitmap. After the header block there is a second general purpose header block, which has the same size as the first header block. The second header can be used for any purposes, for example to store additional information about the file. The second header block is needed for the quadtree and is described later. After the second header block the actual content blocks are stored. These blocks have a fixed length, which is specified at initialization of the buffer manager. The maximum number of data blocks is restricted by the constant `MAXBLOCKS`. Figure 3.2 illustrates the structure of a file. In the example it can be seen, that the first header block contains the number of blocks and the bitmap. The file has four data blocks but only three of them are valid according to the bitmap. Because the size of the bitmap is ten, the maximum number of data blocks this file can have is

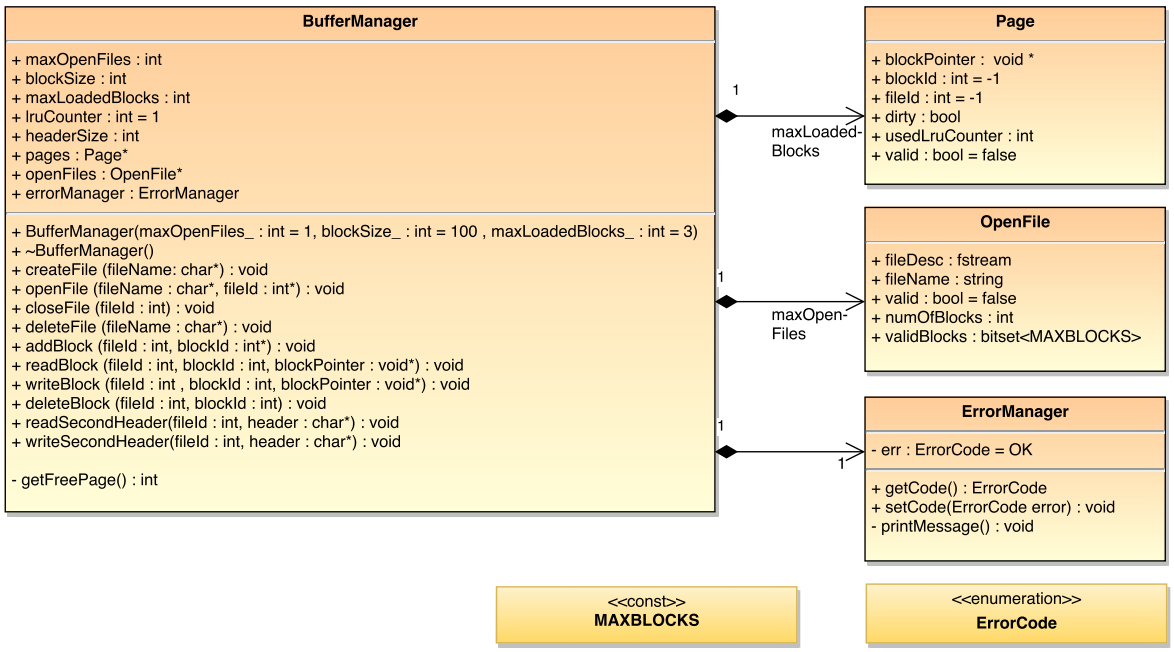


Figure 3.1: Buffer Manager - UML class diagram

ten. After the first header there is the second header which is not used by the buffer manager. Next there are the data blocks one by one.

3.1.2 The class OpenFile

The buffer manager can handle multiple files at the same time. The maximum number of open files can be defined at the initialization of the buffer manager. The class `OpenFile` represents one open file and stores information about that one file. It stores (i) the name of the file in `fileName`, (ii) the file descriptor `fileDesc` of the file provided by the library `<fstream>`, (iii) a boolean called `valid` to know if the file is valid and (iv) the header information of the file; `numOfBlocks` and `validBlocks`. The buffer manager stores an array of `OpenFile` to store information of all open files.

3.1.3 The class Page

The buffer manager can store a predefined amount of blocks. For this purpose, the buffer manager holds an array of the class `Page`. The class `Page` stores information of an in memory loaded block: (i) `blockPointer` pointing to the block in memory; (ii) `fileId` which is the index value of the file to the array `openFiles`; (iii) `blockId` which is the index value of the corresponding block; (iv) boolean `valid` to know if this page is valid; (v) boolean `dirty` to know if the content of the block has been changed in memory but not flushed back to the file; (vi) `usedLRUCounter`, the assigned LRU value of this page.

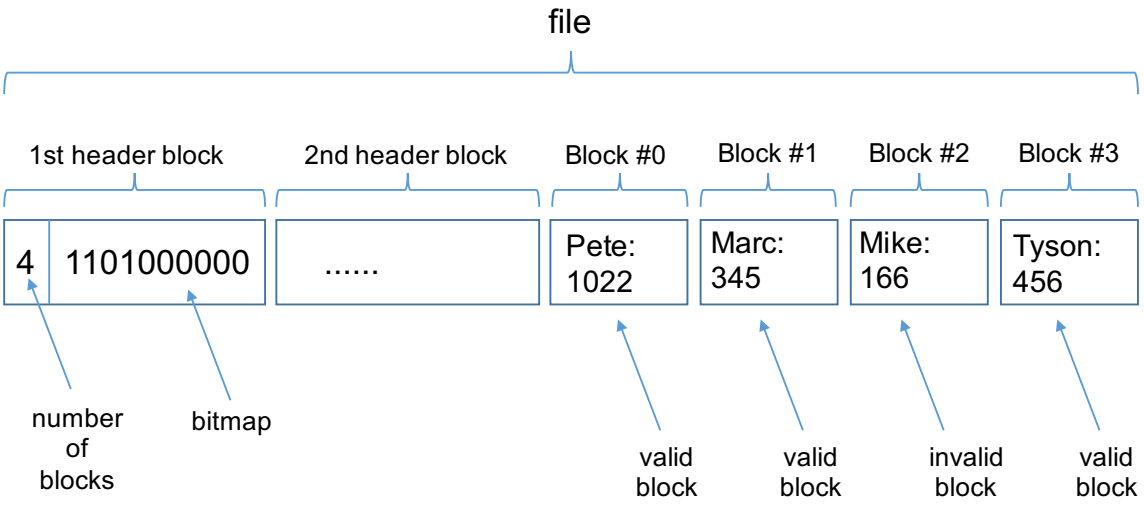


Figure 3.2: Structure of a file

3.1.4 The class ErrorManager

The class `errorManager` is implemented to handle the errors which could occur during run time. It simply holds a variable called `err` of the enumeration type `ErrorCode` to store the current error state. First, `err` is set to the state `OK`. As soon an error occurs `err` is set to the corresponding error state with the setter function by the buffer manager. For example, if a file which does not exist is tried to be opened, the error code `FileNotExists` is set. Then automatically the corresponding error message is printed out to the console. The current error state can be checked by other classes with the getter function. Figure 3.3 shows all the possible error codes.

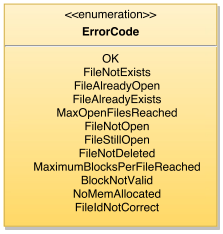


Figure 3.3: Error codes of the buffer manager

3.1.5 The class BufferManager

The class `BufferManager` uses instances of other classes to handle the actual paging. It has (i) `errorManager` for handling the errors, (ii) an array `pages` of the class `Page` and (iii) an array `openFiles` of the class `openFile`. In addition, it holds integers (iv) `maxOpenFiles` and (v) `maxLoadedBlocks` to store the size of `openFiles` and `pages`. Also there is (vi) a parameter for storing the size of blocks called `blockSize`. The (vii)

counter `lruCounter` is the global LRU counter and initialized with 1. Also (viii) the size of the header blocks in bytes is stored in a parameter called `headerSize`. `headerSize` is calculated in the constructor.
The following functions are provided by the class `BufferManager` as public functions: (i) a constructor, which initializes the different size parameters; functions to (ii) create a binary file readable for the buffer manager, (iii) to open an existing file, (iv) to close an open file and (v) to delete a file; and functions (vi) to read a block, (vii) to add a block and (viii) to write a block. There are also a (ix) getter and (x) setter function to access the second header of the file (`readSecondHeader`; `writeSecondHeader`), which can be used by other classes. All these functions are described in detail in the following sections.

The constructor

The constructor of the buffer manager takes three input arguments that are used to initialize (i) `maxOpenFiles`, (ii) `blockSize` and (iii) `maxLoadedBlocks`. All three parameters have a default value. The input arguments are checked and if they have an invalid value, they are set to the default values. After this, `headerSize` is calculated, so the header is big enough to store all necessary information. At the end the arrays `pages` and `openFiles` are initialized and the memory space for the pages is allocated.

```
BufferManager::BufferManager(int maxOpenFiles_, int blockSize_ , int
    maxLoadedBlocks_): maxOpenFiles(maxOpenFiles_), blockSize(blockSize_)
    , maxLoadedBlocks(maxLoadedBlocks_) {
     if (maxOpenFiles < 1) maxOpenFiles = 1;
     if (blockSize < 100) blockSize = 100;
     if (maxLoadedBlocks < 3) blockSize = 3;

     //Calculate size of header
     headerSize = 3* sizeof(int) + sizeof(bitset<MAXBLOCKS>);

     //Initialize arrays
     pages = new Page[maxLoadedBlocks];
     openFiles = new OpenFile[maxOpenFiles];

     //Allocate memory for pages
     for (int i = 0; i < maxLoadedBlocks; i++) {
         pages[i].blockPointer = malloc(blockSize);
     }
};
```

createFile

The function `createFile` creates a binary file, which can be read and written by the buffer manager. It takes one input argument `fileName` for the name of the file. This function only creates the file. To use this newly created file with the buffer manager the file has to be opened after being created.
When this function is called, at first the buffer manager checks if there is already a file with this name. If so the corresponding error code is set in `errorManager`, which then automatically

prints an error message to the console, and the function stops.
If no error occurs the file is created, the number of blocks and the bitmap (both initialized with zeros) are written to the file using the <fstream> library. After that the file is closed.

```
void BufferManager::createFile (char *fileName){

        //Check if file exists already
        if(fileExists(fileName)){
            errorManager.setCode(FileAlreadyExists);
            return;
        }
        //Create file
        fstream newfile;
        newfile.open(fileName, ios::binary | ios::out);

        //Write header information to a new file
        int headerNumberOfBlocks = 0;
        bitset<MAXBLOCKS> headerBitMap;
        newfile.write((char*)&headerNumberOfBlocks, sizeof(int));
        newfile.write((char*)&headerBitMap, sizeof(bitset<MAXBLOCKS>));

        newfile.close();
    };
```

openFile

To open a file with the buffer manager the function `openFile` is implemented. The first input arguments `fileName` identifies the file, the second argument `fileId` is a pointer to an integer. This pointer is used to return the index value of the file in `openFiles`, which is needed for further actions to refer to the correct open file.
When the function is called, the buffer manager checks if a file with the name exists and if it is already open by the buffer manager. If everything is ok, an invalid `OpenFile` in the `openFiles` array is searched. After one has been found the file is opened. Then all `OpenFile` variables are updated: `fileName` is assigned and the header of the file is copied to `numberOfBlocks` and `validBlocks`. At the end the `OpenFile` variable `valid` is set to 1 and `fileId` is returned by assigning the position of the file in the `openFiles` array to the argument `fileId`.

```
void BufferManager::openFile (char *fileName, int *fileId){

        //Check if file exists
        if (!fileExists(fileName)) {
            errorManager.setCode(FileNotExists);
            *fileId = -1;
            return;
        }
        //Check if file is already open
        if (findOpenFile(fileName) != -1) {
            errorManager.setCode(FileAlreadyOpen);
            *fileId = findOpenFile(fileName);
```

```
            return;
        }
        //Find empty entry in openFiles and load file information to
            openFiles
        int i = findEmptyOpenFile();
        if (i == -1) {
            errorManager.setCode(MaxOpenFilesReached);
            return;
        }
        openFiles[i].fileDesc.open(fileName, ios::binary | ios::in | ios
            ::out);

        openFiles[i].fileName = fileName;
        openFiles[i].fileDesc.read((char*)&openFiles[i].numOfBlocks,
            sizeof(int));
        openFiles[i].fileDesc.read((char*)&openFiles[i].validBlocks,
            sizeof(bitset<MAXBLOCKS>));
        openFiles[i].valid = 1;

        *fileId = i;
    };
```

addBlock

The function `addBlock` is implemented to add a new block to an open file. It adds a new empty block to the file. This function has two input arguments, (i) `fileId` and (ii) `blockId`. `fileId` is needed to refer to the correct file, as there can be more than one open files. `blockId`, a pointer to an integer, is used to return the index value of the new block in the file. Only using the correct id, the block can later be accessed.
First, this function checks if `fileId` is valid. Then an invalid block in the file is searched in `validBlocks`. If for this a new block is added at the end of the file, the counter `numberOfBlocks` is increased by 1. The corresponding bit in `validBlocks` is set to 1 to mark this block as valid. Then a page is requested. The function `getFreePage` returns the position of an empty page in `pages`. This function is explained in more detail later. Then the page variables are updated: `blockId`, `fileId` and `valid`. The variable `dirty` is set to 1, so even if the block stays empty it is eventually written to the file and can later be accessed without reading beyond the end of the file. The value of `lruCounter` is assigned to `usedLruCounter` of the page. The global variable `lruCounter` is then increased by one.

```
void BufferManager::addBlock (int fileId, int *blockId){

        //Check if fileId is valid
        if (fileId < 0 || fileId > maxOpenFiles || openFiles[fileId].
            valid == false) {
            errorManager.setCode(FileIdNotCorrect);
            *blockId = -1;
            return;
        }
```

```
        //Find empty block in file
        int i = findEmptyBlock(fileId);
        if (i == -1) {
            errorManager.setCode(MaximumBlocksPerFileReached);
            *blockId = -1;
            return;
        }
        //Update numOfBlocks
        if (i + 1 > openFiles[fileId].numOfBlocks) {
            openFiles[fileId].numOfBlocks++;
        }
        openFiles[fileId].validBlocks[i] = 1;
        *blockId = i;
        //Create Page
        int pageId = getFreePage();
        pages[pageId].blockId = i;
        pages[pageId].fileId = fileId;
        pages[pageId].dirty = 1;
        pages[pageId].valid = 1;
        pages[pageId].usedLruCounter = lruCounter++;
    };
```

readBlock

The function `readBlock` provides the function to read data from a block. This function has three input arguments; (i) `fileId`, (ii) `blockId` and (iii) `blockPointer`. `fileId` and `blockId` are needed to identify and locate the requested block. Input argument `blockPointer` should point to separate memory. The requested block is then copied to the memory pointed by the `blockPointer`. That means the read requestor has to assure to allocate memory of at least `blockSize`. By returning only a copy of the block, the requestor can not make changes directly to the buffered blocks without using the `writeBlock` function provided by the buffer manager. This way the page variable `dirty` is always correct. First, `fileId` and `blockId` is checked. After that, it is checked if the requested block is already loaded in `pages`. If so, the block of the page is copied to the memory pointed by the `blockPointer` of the input argument and the `usedLruCounter` of that page is updated. If not, a page is requested with the `getFreePage` function. The variables of the empty page are updated the same way like in the `addBlock` function except the variable `dirty`, this is set to 0, since the block in the file and in the page do not differ. Then the block in the file is copied to the memory of the page. For this, the file is read from the beginning of the requested block. After this, the block is copied to the memory pointed by `blockPointer` of the input argument.

```
    void BufferManager::readBlock (int fileId, int blockId, void *
        blockPointer){

        //Check if fileId is valid
        if (fileId < 0 || fileId > maxOpenFiles || openFiles[fileId].
            valid == false) {
            errorManager.setCode(FileIdNotCorrect);
```

```
            return;
        }
        //Check if block is valid
        if (blockId < 0 || blockId >= openFiles[fileId].numOfBlocks ||
            openFiles[fileId].validBlocks[blockId] == 0) {
            errorManager.setCode(BlockNotValid);
            return;
        }
        //Check if block is already loaded
        for (int i = 0; i < maxLoadedBlocks; i++) {
            if (pages[i].valid == true && pages[i].fileId == fileId &&
                pages[i].blockId == blockId) {
                pages[i].usedLruCounter = lruCounter++;
                if ((int*)blockPointer == NULL) {
                    errorManager.setCode(NoMemAllocated);
                    return;
                }
                memcpy(blockPointer, pages[i].blockPointer, blockSize);
                return;
            }
        }
        //Create Page if block is not already loaded
        int pageId = getFreePage();
        pages[pageId].blockId = blockId;
        pages[pageId].fileId = fileId;
        pages[pageId].dirty = 0;
        pages[pageId].valid = 1;
        pages[pageId].usedLruCounter = lruCounter++;
        openFiles[fileId].fileDesc.seekg(2 * headerSize + blockId *
            blockSize);
        openFiles[fileId].fileDesc.read((char*)pages[pageId].
            blockPointer, blockSize);

        if (blockPointer == NULL) {
            errorManager.setCode(NoMemAllocated);
            return;
        }
        memcpy(blockPointer, pages[pageId].blockPointer, blockSize);
    };
```

writeBlock

The buffer manager provides the function `writeBlock` to modify an existing block. This function takes the same three input arguments as the function `readBlock`; (i) `fileId`, (ii) `blockId` and (iii) `blockPointer`. `fileId` and `blockId` are used to identify the correct block. The requestor has to pass `blockPointer` which points to the modified block. By passing a pointer to a separate block in memory it can be ensured that the requestor can not modify the blocks directly without using the provided functions.
When the function is called `fileId` and `blockId` are checked. Then the requested block is searched in `pages`. If the block has not been found a new page is requested with the function

getFreePage. So in both cases a page is available. Next the page variables fileId and blockId are updated. Then the modified block is copied to the memory of the page. Finally, usedLruCounter is updated and the variables dirty and valid is set to 1.

```
void BufferManager::writeBlock (int fileId, int blockId, void *
    blockPointer){
      //Check if fileId is valid
      if (fileId < 0 || fileId > maxOpenFiles || openFiles[fileId].
          valid == false) {
          errorManager.setCode(FileIdNotCorrect);
          return;
      }
      // Check if block is valid
      if (blockId < 0 || blockId >= openFiles[fileId].numOfBlocks ||
          openFiles[fileId].validBlocks[blockId] == 0) {
          errorManager.setCode(BlockNotValid);
          return;
      }
      // Check if block is already loaded
      int pageId = -1;
      for (int i = 0; i < maxLoadedBlocks; i++) {
          if (pages[i].fileId == fileId && pages[i].blockId == blockId
              && pages[i].valid == true) {
              pageId = i;
          }
      }
      // If block is not already loaded, get empty Page
      if (pageId == -1) {
          pageId = getFreePage();
      }
      // Fill Page with information and block-content
      pages[pageId].blockId = blockId;
      pages[pageId].fileId = fileId;
      memcpy(pages[pageId].blockPointer, blockPointer, blockSize);
      pages[pageId].usedLruCounter = lruCounter++;
      pages[pageId].dirty = 1;
      pages[pageId].valid = 1;
  };
```

getFreePage

The function getFreePage returns the position of an empty page in pages. First, pages is searched for an invalid page. If one has been found, the index value of this page is returned. If no invalid page has been found, the page with the lowest usedLruCounter is searched. As soon this page has been found, it is emptied. If the variable dirty is 1, the block is written to the file in disk. Next, valid and dirty are set to 0. Finally, the index value of the page is returned.

```
int BufferManager::getFreePage(){
      //If empty Page exists return this PageId
      for(int i = 0; i < maxLoadedBlocks; i++){
```

```
            if (pages[i].valid == false) {
                return i;
            }
        }
        //Find Page with lowest usedLruCounter and clear Page
        int minimum = pages[0].usedLruCounter;
        int pageId = 0;

        for(int i = 0; i < maxLoadedBlocks; i++){
            if (pages[i].usedLruCounter < minimum) {
                minimum = pages[i].usedLruCounter;
                pageId = i;
            }
        }
        if (pages[pageId].dirty == true)
            loadBlockToDisk(pages[pageId].fileId, pages[pageId].blockId,
                pages[pageId].blockPointer);

        pages[pageId].valid = false;
        pages[pageId].dirty = false;

        return pageId;
    };
```

deleteBlock

The function `deleteBlock` deletes the block that is identified by the input arguments `fileId` and `blockId`. First, these two arguments are checked for reasonable values. Then the block is searched in `pages`. If it has been found, the page is set to invalid. At the end the block is set to invalid in `validBlocks` of the `OpenFile`. The block in the file is not deleted explicitly to reduce I/O operations. When a new bock is added a previously deleted block is used.

```
void BufferManager::deleteBlock(int fileId, int blockId){

        //Check if fileId is valid
        if (fileId < 0 || fileId > maxOpenFiles || openFiles[fileId].
            valid == false) {
            errorManager.setCode(FileIdNotCorrect);
            return;
        }
        // Check if block is valid
        if (blockId < 0 || blockId >= openFiles[fileId].numOfBlocks ||
            openFiles[fileId].validBlocks[blockId] == 0) {
            errorManager.setCode(BlockNotValid);
            return;
        }
        //Search block in pages and set invalid
        for (int i = 0; i < maxLoadedBlocks; i++) {
            if (pages[i].fileId == fileId && pages[i].blockId == blockId
                )
```

```
                pages[i].valid = 0;
            }
            openFiles[fileId].validBlocks[blockId] = 0;
        };
```

closeFile

This function closes the file identified with `fileId` passed by the input arguments. First, it is checked if the file with `fileId` is valid. In that case, one by one all pages of that file in `pages` are set invalid. If a page is dirty the block is written to the disk. Then the variable `valid` is set to 0. After all pages of this file have been set invalid, the header of the file from `openFiles` is written to the file in disk and finally the open file is set to 0 and the file is closed.
The destructor of `BufferManager` uses also this function to close all open files, so all dirty pages are eventually written to the disk even without explicitly calling the `closeFile` function.

```
void BufferManager::closeFile (int fileId){
        //Check if file is open
        if (openFiles[fileId].valid == false){
            errorManager.setCode(FileNotOpen);
            return;
        }
        //Find file, load blocks & header to disk and close file
        for (int i = 0; i < maxLoadedBlocks; i++) {
            if (pages[i].fileId == fileId && pages[i].valid == true) {
                if (pages[i].dirty == true)
                    loadBlockToDisk(fileId, pages[i].blockId, pages[i].
                        blockPointer);
                pages[i].valid = 0;
            }
        }
        loadHeaderToDisk(fileId);
        openFiles[fileId].valid = 0;
        openFiles[fileId].fileDesc.close();
    };
```

deleteFile

This function deletes a closed file from the disk. If the file is still open in the buffer manager, then an error is reported. If the file is closed, it is deleted with the standard function `remove` and then checked for errors.

```
void BufferManager::deleteFile (char *fileName){
        //Check if file is open
        for (int i = 0; i < maxOpenFiles; i++) {
            if (openFiles[i].valid == true && strcmp(openFiles[i].
                fileName.c_str(), fileName)) {
                errorManager.setCode(FileStillOpen);
                return;
```

```
                }
            }
            //Delete and check if file is deleted
            if( remove(fileName) != 0 )
                errorManager.setCode(FileNotDeleted);
        };
```

3.2 Quadtree

The quadtree is able to create files, that represent point quadtrees, and to insert and search points in them using the buffer manager. A point, represented by the class `Point`, that has two fields of type floating point: `x` and `y`. For every inserted point an instance of `QTreeNode`, representing a node, is created that stores the point and also five `NodePointers`; four for each child and one for the parent. The nodes are stored in blocks and the maximum number of nodes per block is restricted by the constant `MAXNODESPERBLOCK`. `NodePointer` is used to identify a node in the file. `NodePointer` has two fields: (i) `blockId` identifies the block where the node is stored and (ii) `nodeId` identifies the position of the node in the block.
Five main classes have been implemented to provide the main functionality of the quadtree: (i) `QTree`, (ii) `QTreeHeader`, (iii) `BufferManager`, (iv) `QErrorManager` and (v) `OpenScan`. The class diagram is illustrated in 3.4.

3.2.1 The file structure

To create a quadtree file, first a file is created by `BufferManager`. As described before such a file starts with a header block used by the buffer manager. After that there is a second header block which is used by the quadtree to store additional header information about the quadtree; namely `numberOfPoints`, `root` and a bitmap `fullBlocks`. In the following data blocks the quadtree nodes are stored. Each of these blocks start with a bitmap called `blockHeader`, which stores information about which nodes in the block are valid. After the bitmap the nodes are stored one by one. Up to `MAXNODESPERBLOCK` nodes can be stored in each block. Figure 3.5 illustrates a quadtree file. The file starts with the first header block used by the buffer manager. In the second header block the additional information for the quadtree are stored. It can be seen, that three points are stored in the quadtree file. The root of the quadtree is identified by the nodepointer $\begin{smallmatrix}0\\1\end{smallmatrix}$ meaning the root is stored in `Block#0` at position `Node#1`. In the bitmap `fullblocks` it can bee seen, that `Block#1` is not full, all the other blocks are either full or invalid. A data block starts with bitmap `blockHeader` to know which nodes are valid in this block. Then the nodes are stored. First, the nodepointer to the parent then to the four children are stored. If the node has no child or parent, it is indicated with `blockId -1`. After the nodepointers, the point $\begin{smallmatrix}x\\y\end{smallmatrix}$ is stored.

3.2.2 The class QTreeHeader

This class stores information about one open quadtree file additionally to `OpenFile` of the buffer manager. It has three fields (i) `fullBlocks`, a bitmap to know if a block is full, (ii)

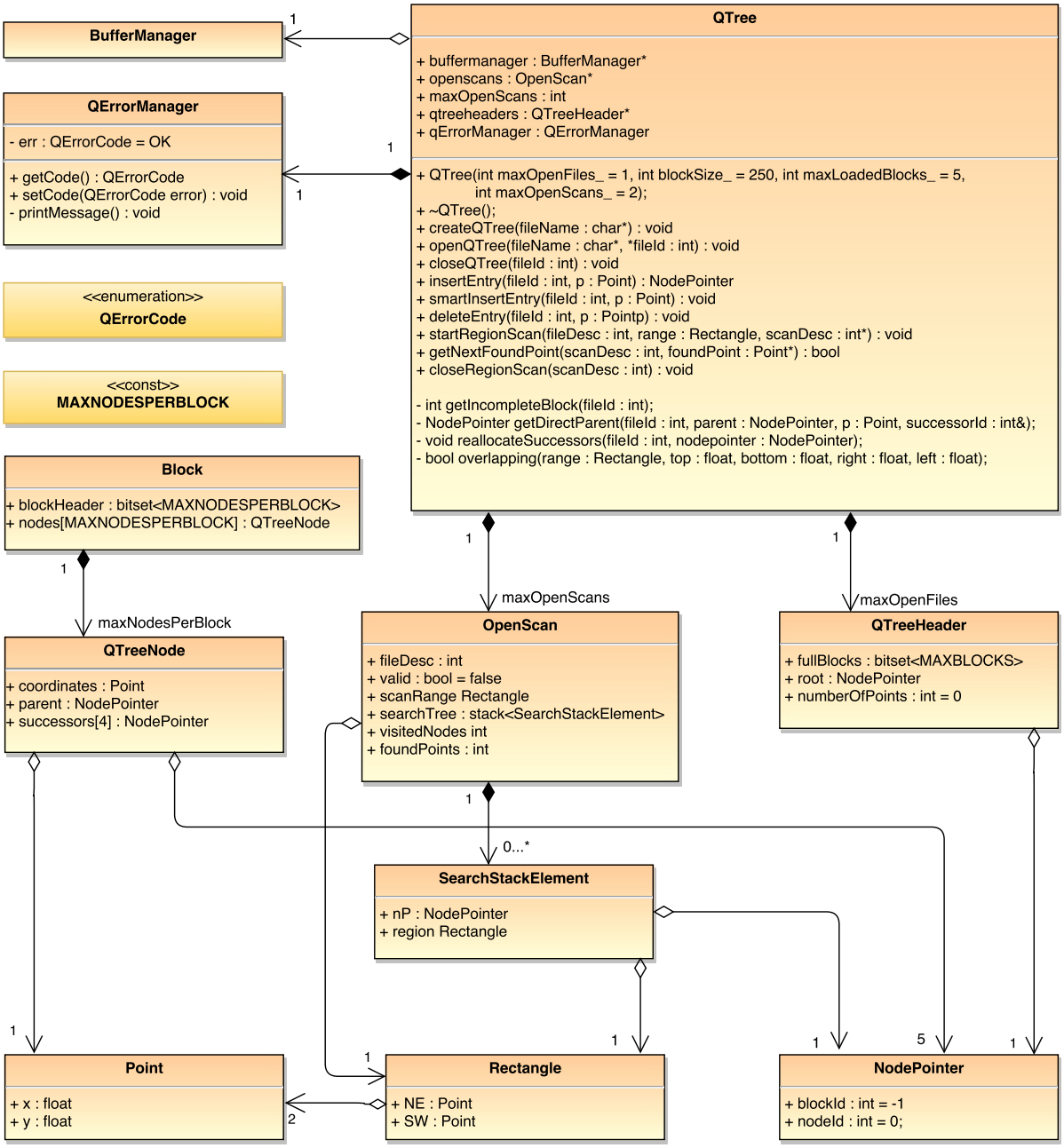


Figure 3.4: Quadtree - UML class diagram

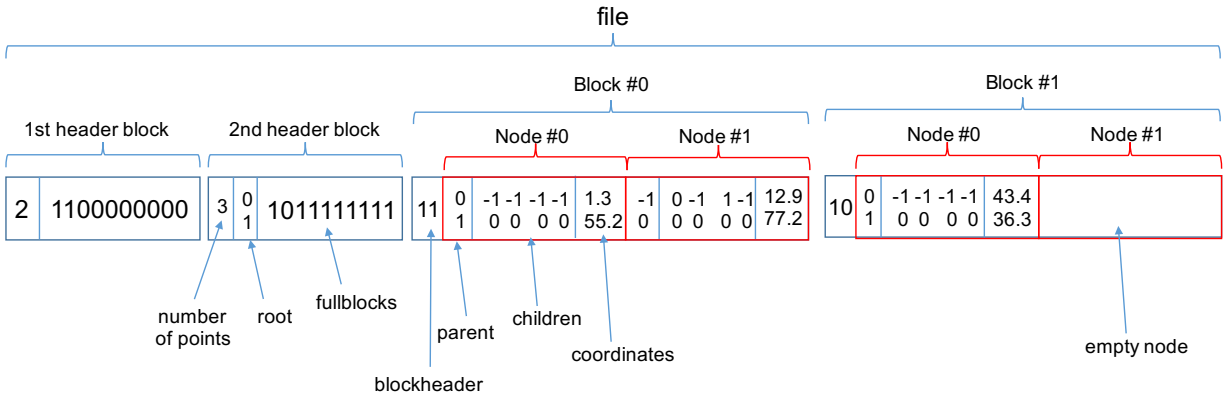


Figure 3.5: File structure of a quadtree file

`root`, a nodepointer to the root and (iii) `numberOfPoints` counting the number of points inserted in this quadtree. The bitmap `fullBlocks` is used to know if a new node can be inserted into a block: 0 means the block is not full and valid and can be used to insert a new node; 1 means the block is either full or invalid so this block can not be used directly. Every quadtree file stores its `QTreeHeader` in the second header block of the file.

3.2.3 The class QErrorManager

This class has been implemented to handle the errors of the quadtree. It works exactly the same way as `ErrorManager` of the buffer manager but with another enumeration type called `QErrorCode` as the current error state. Figure 3.6 shows all the possible error codes of the quadtree.

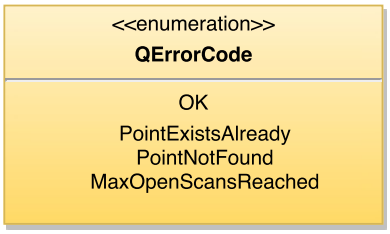


Figure 3.6: Error codes of the quadtree

3.2.4 The class OpenScan

The class `OpenScan` is implemented to handle a search with a search region. It represents one open search. This class is used to return found points one by one. Since the goal of this quadtree is to work with files that do not fit in memory, it would not make sense to load all the found points to memory at once.
The search region is a rectangle, represented by the class `Rectangle`, that is defined by

two Point values (SW and NE). The class OpenScan has the following input arguments: (i) fileDesc, to identify the file the search belongs to, (ii) valid, a boolean to know if it is still open, (iii) scanRange as the search region, (iv) searchTree, (v) visitedNodes, an integer for the number of visited nodes and (vi) foundPoints, an integer for the number of found points. To track which nodes have to be checked next, the searchTree, a stack, stores the nodepointers of the nodes to be checked next and the search region for this check. Using this stack, the search does not need to start from the beginning after one point has been found, but can continue with the remaining nodes. More details about how the region search is implemented are given later, in the subsections where the functions startRegionScan and getNextFoundPoint are described.

3.2.5 The class QTree

The class QTree uses instances of the aforementioned classes to provide all the main functions for the quadtree. The class has: (i) buffermanager, an instance of BufferManager, (ii) qtreeheaders, a QTreeHeader array of size maxOpenFiles, (iii) an integer maxOpenScans, (iv) openscans, an OpenScan array of size maxOpenScans and (vi) qErrorManager, an instance of QErrorManager. The following functions are provided by the QTree as public functions: (i) a constructor to initialize all parameters; a function (ii) to create a quadtree file, (iii) to open a quadtree file, (iv) to close a file, (v) to insert a point, (v) to insert a point with balancing afterwards, (vi) to delete a point, (vii) to open and initialize a search, (vii) to get the next found point of a search and (viii) to close an open search. These functions are described in detail in the following subsections.

The constructor

The constructor of QTree takes four input arguments. The first three are the same as in constructor of BufferManager and they are used to initialize buffermanager. The forth argument is used to initialize maxOpenScans. If the forth argument has a value smaller than 1, maxOpenScans is set to 1. The arrays openscans and qtreeheaders are initialized with sizes maxOpenScans and maxOpenFiles.

```
QTree::QTree(int maxOpenFiles_, int blockSize_ , int maxLoadedBlocks_,
    int maxOpenScans_){
     buffermanager = new BufferManager(maxOpenFiles_, blockSize_ ,
         maxLoadedBlocks_);
     if (maxOpenScans_ > 0)
          maxOpenScans = maxOpenScans_;
     else
          maxOpenScans = 1;
     openscans = new OpenScan[maxOpenScans];
     qtreeheaders = new QTreeHeader[buffermanager->maxOpenFiles];
};
```

createQTree

This function creates a quadtree file with the name given as input argument. First, a file is created by the buffer manager. If no errors occur, an instance of `QTreeHeader` is written to the second header of the file. `fullBlocks` is first filled with 1s since at the beginning all data blocks are invalid.

```
void QTree::createQTree(char* fileName){
    //Checking for errors
    buffermanager->errorManager.setCode(OK);
    buffermanager->createFile(fileName);
    if (buffermanager->errorManager.getCode()!= OK)
        return;
    //Create QTree-header with bitmap (fullBlocks) all 1's
    QTreeHeader header;
    for (int i = 0; i < MAXBLOCKS; i++) {
        header.fullBlocks[i] = 1;
    }
    //Open file and add QTree-header
    int fileId;
    buffermanager->openFile(fileName, &fileId);
    if (buffermanager->errorManager.getCode()!= OK)
        return;
    buffermanager->writeSecondHeader(fileId, (char*) &header);
    buffermanager->closeFile(fileId);
};
```

openQTree

The function `openQTree` is implemented to open a quadtree file and load all the relevant header information of this file. First, the file with the name `fileName` is opened with the buffer manager. After that, the second header block of the file is copied to `qtreeheaders` at index position `fileId`.

```
void QTree::openQTree(char *fileName, int *fileId){
    //Checking for errors
    buffermanager->errorManager.setCode(OK);
    buffermanager->openFile(fileName, fileId);
    if (buffermanager->errorManager.getCode()!= OK)
        return;
    //Copying QTree-header
    void* header = malloc(buffermanager->headerSize);
    buffermanager->readSecondHeader(*fileId, (char*) header);
    memcpy((void*) &qtreeheaders[*fileId], header, sizeof(QTreeHeader));
    free(header);
};
```

closeQTree

This function closes an open quadtree file. First, it checks if the file identified by `fileId` is open. If not, an error is set by the buffer manager. After the check the `QTreeHeader` of that

file is written to the second header of the file and then closed by the buffer manager.

```
void QTree::closeQTree(int fileId){
        //Check if file is open
    if (buffermanager->openFiles[fileId].valid == false){
        buffermanager->errorManager.setCode(FileNotOpen);
        return;
    }
    //flush QTree-header to disk and close file
    buffermanager->writeSecondHeader(fileId, (char*) &qtreeheaders[
        fileId]);
    buffermanager->closeFile(fileId);
};
```

insertEntry

This function inserts a point into an open quadtree. It has two input arguments; `fileId` and point `p`. The insertion has four main steps. (i) First, the block, in which the point is stored later, is copied to memory. (ii) Into this copy the new point is inserted and the nodepointer to the parent is updated. (iii) Then a write of this edited copy is requested. (iv) And at the end the parent node is updated with its new child. In the following these steps are explained in more detail.
First, a nodepointer `nP` is created pointing to the position in the file where the new node is inserted later. For this, a block is searched which has at least one free place for a new node. The function `getIncompleteBlock` searches for a 0 in `fullBlocks`, that means the block is valid and not full. If no block is found, this function adds a new block to the file with the buffer manager. The returned value is saved as `blockId` of `nP`. Then the block identified with `blockId` is copied to `block` in memory. An empty position is searched in `block`, which is then saved as `nodeId` of `nP`.
Next the header of `block` is updated since a new node is being inserted. The point `p` is copied into the empty node. Before loading `block` back to the file the parent of this new node is determined. If the quadtree is empty, the new node is set as root of the quadtree. Since then the new node has no parent node, the nodepointer to the parent remains a null pointer (`blockId` = -1). If the quadtree is not empty, the appropriate parent is searched. This is done with the function `getDirectParent`, which recursively searches an empty child node in the quadtree. The nodepointer to the parent of the new node in `block` is then updated.
Then a write of `block` is requested using `writeBlock`.
After that the parent node is updated with its new child node, by calling `addSuccessorToParent`. At the end `numberOfPoints` is increased by 1 and the `NodePointer` of the new node is returned.

```
NodePointer QTree::insertEntry(int fileId, Point p){
    //Check if file is open
    if (buffermanager->openFiles[fileId].valid == false){
        buffermanager->errorManager.setCode(FileNotOpen);
        return;
        }
    //Build nodepointer for the new point
```

```
    NodePointer nP;
    nP.blockId = getIncompleteBlock(fileId);
    if (nP.blockId == -1){
        buffermanager->errorManager.setCode(MaximumBlocksPerFileReached)
            ;
        return NodePointer();
        }
    //Find free nodeId
    void* blockPointer = malloc(buffermanager->blockSize);
    buffermanager->readBlock(fileId, nP.blockId, blockPointer);
    Block* block = (Block*) blockPointer;
    for (int i = MAXNODESPERBLOCK - 1; i > -1; i--) {
        if (block->blockHeader[i] == 0)
            nP.nodeId = i;
    }
    //Update block-data
    block->blockHeader[nP.nodeId] = 1;
    if(blockIsFull(block->blockHeader))
        qtreeheaders[fileId].fullBlocks[nP.blockId] = 1;
    block->nodes[nP.nodeId].coordinates = p;
    //If there is no root, set the new node as root
    if (qtreeheaders[fileId].root.blockId == -1){
        qtreeheaders[fileId].root.blockId = nP.blockId;
        qtreeheaders[fileId].root.nodeId = nP.nodeId;
        buffermanager->writeBlock(fileId, nP.blockId, (void*) block);
    } else{
        //Else find parent and write node to file plus update parent-
            node
        int successorId;
        NodePointer parentPointer = getDirectParent(fileId, qtreeheaders
            [fileId].root, p, successorId);
        //Errormessage if point exists already
        if (parentPointer.blockId == -1) {
            qErrorManager.setCode(PointExistsAlready);
            return nP;
        }
        //Update parent and write to file
        block->nodes[nP.nodeId].parent = parentPointer;
        buffermanager->writeBlock(fileId, nP.blockId, (void*) block);
        //Update the parent node
        addSuccessorToParent(fileId, parentPointer, nP, successorId);
    }
    free(blockPointer);
    qtreeheaders[fileId].numberOfPoints++;
    return nP;
};
```

smartInsertEntry

The function `smartInsertEntry` inserts a point into a quadtree and balances the new leaf like proposed by Finkel and Bentley [FB74]. It has the same two input arguments as

insertEntry. First, the point is inserted using insertEntry. After it has been inserted without errors, it is checked if the conditions for a balancing are given. For this the new node (firstNode), its parent (secondNode) and the parent of the parent (thirdNode) are loaded to memory. Then secondNode and thirdNode are checked for the number of child nodes. The balancing can only be applied if both have only one child. When checking the number of children, the index value of the child is also stored in firstSuccessorId and secondSuccessorId. These are needed to determine later which type of balancing should be applied.
To apply the balancing, first the three nodes are separated from the quadtree by deleting the corresponding child nodepointers. Then in a specific sequence the three nodes are reinserted to the quadtree using the function reallocateSuccessors. This function reallocateSuccessors basically edits only the nodepointers (to parent and to child) to connect the node again with the quadtree. The order in which these three nodes are reinserted depends if a double or a single balance is applied. To determine this, firstSuccessorId and secondSuccessorId are checked with the function conjugates. This function compares these two integers and since they represent the four intercardinal directions (NE, NW, SW, SE) they can be used to check if firstNode lays in the rectangle set up by the other two nodes. If this is the case, double balance is applied, by first reinserting firstNode and then the other two. If this is not the case, single balance is applied by first reinserting secondNode and then the other two.

```
void QTree::smartInsertEntry(int fileId, Point p){
    //Insert point and check for errors
    buffermanager->errorManager.setCode(OK);
    NodePointer firstNP = insertEntry(fileId,p);
    if (buffermanager->errorManager.getCode()!= OK || firstNP.blockId ==
        -1)
        return;
    //Access the new node and access the parent & parent of parent
    QTreeNode firstNode = getNode(fileId, firstNP);
    if (firstNode.parent.blockId == -1) {
        return;
    }
    NodePointer secondNP = firstNode.parent;
    QTreeNode secondNode = getNode(fileId, secondNP);
    if (secondNode.parent.blockId == -1) {
        return;
    }
    NodePointer thirdNP = secondNode.parent;
    QTreeNode thirdNode = getNode(fileId, thirdNP);
    //Check how many children the parents do have
    //And find position of child in successors[] of parent
    int successorsCount = 0;
    int firstSuccessorId = -1;
    int secondSuccessorId = -2;
    for (int i = 0; i < 4; i++) {
        if(secondNode.successors[i].blockId != -1)
            successorsCount++;
```

```
            if (secondNode.successors[i].blockId == firstNP.blockId &&
                secondNode.successors[i].nodeId == firstNP.nodeId)
                 firstSuccessorId = i;
            if(thirdNode.successors[i].blockId != -1)
                 successorsCount++;
            if (thirdNode.successors[i].blockId == secondNP.blockId &&
                thirdNode.successors[i].nodeId == secondNP.nodeId)
                 secondSuccessorId = i;
        }
        //If both parents have only one child, continue
        if (successorsCount != 2)
            return;
        //Separate the three nodes from the QTree
        //If thirdnode = root, delete root
        if (thirdNode.parent.blockId == -1) {
            qtreeheaders->root = NodePointer();
        } else{
            deleteChildFromParent(fileId, thirdNode.parent, thirdNP);
        }
        deleteChildFromParent(fileId, thirdNP, secondNP);
        deleteChildFromParent(fileId, secondNP, firstNP);
        //First case: double balance
        if (conjugates(firstSuccessorId, secondSuccessorId)){
            reallocateSuccessors(fileId, firstNP);
            reallocateSuccessors(fileId, secondNP);
            reallocateSuccessors(fileId, thirdNP);
        }
        //Second case: single balance
        else {
            reallocateSuccessors(fileId, secondNP);
            reallocateSuccessors(fileId, firstNP);
            reallocateSuccessors(fileId, thirdNP);
        }
};
```

deleteEntry

This function provides the option to delete a node in a quadtree file. It has two input arguments, (i) `fileId` and (ii) `p`, the point to delete. `p` is searched in the quadtree. As soon as it has been found, the node is loaded to `node` in memory. Next the corresponding child nodepointer of the parent is deleted, so this node is separated from the quadtree. Next the whole block, which contains the node, is loaded to memory. In the header of the block the corresponding node is set to invalid and all nodepointers of the node are deleted. If the block was full before deleting the node, the block is marked as not full in `fullBlocks` of `qtreeheaders`. The modified block is then loaded back to the file. After that, all child nodes of this deleted node are reinserted recursively into the quadtree. Even though the node has been deleted, all its nodepointers are still accessible with the variable `node`. At the end the variable `numberOfPoints` is decreased by 1.

```
void QTree::deleteEntry(int fileId, Point p){
```

```
    //Check if file is open
    if (buffermanager->openFiles[fileId].valid == false){
        buffermanager->errorManager.setCode(FileNotOpen);
        return;
        }
    //Find node with the point
    NodePointer nodePointer;
    if (!findNode(fileId, qtreeheaders[fileId].root, p, nodePointer)) {
        qErrorManager.setCode(PointNotFound);
        return;
    }
    QTreeNode node = getNode(fileId, nodePointer);
    //Delete child from parent
    deleteChildFromParent(fileId, node.parent, nodePointer);
    //Load block with node
    void* blockPointer = malloc(buffermanager->blockSize);
    buffermanager->readBlock(fileId, nodePointer.blockId, blockPointer);
    Block* block = (Block*) blockPointer;
    //Delete the node (header-bitmap, parent, successors)
    buffermanager->readBlock(fileId, nodePointer.blockId, (void*) block)
        ;
    block->blockHeader[nodePointer.nodeId] = 0;
    block->nodes[nodePointer.nodeId].parent.blockId = -1;
    for (int i = 0; i < 4; i++) {
        block->nodes[nodePointer.nodeId].successors[i].blockId = -1;
    }
    //Update the blockheader
    if (!blockIsFull(block->blockHeader))
        qtreeheaders[fileId].fullBlocks[nodePointer.blockId] = 0;
    buffermanager->writeBlock(fileId, nodePointer.blockId, (void*) block
        );
    //reallocate successors recursively
    srand ((int)time(0));
    int random = rand();
    for (int i = 0; i < 4; i++) {
        reallocateSuccessors(fileId, node.successors[(random + i) %4]);
    }
    free(blockPointer);
    qtreeheaders[fileId].numberOfPoints--;
};
```

startRegionScan

This function only initializes an `OpenScan` for a new search but does not search for points. The input arguments are (i) `fileId` of the quadtree file, (ii) `range`, the rectangle to be searched and (iii) `scanId`. `scanId` is used to return the index value in `openscans` used to store the new scan. First, an invalid `OpenScan` is searched in `openscans` and assigned to `scanId`. If one has been found, all `OpenScan` variables are initialized. `searchTree` is used to know the order in which the tree is searched next. At initialization, the root and `range` are inserted into `searchTree`. Hence the search starts with the root.

```
void QTree::startRegionScan(int fileId, Rectangle range, int* scanId){
    //Find emty OpenScan
    *scanId = findEmpyScan();
    if (*scanId == -1) {
        qErrorManager.setCode(MaxOpenScansReached);
        return;
    }
    //Init OpenScan variables
    openscans[*scanId].valid = true;
    openscans[*scanId].fileDesc = fileId;
    openscans[*scanId].scanRange = range;
    openscans[*scanId].visitedNodes = 0;
    openscans[*scanId].foundPoints = 0;

    SearchStackElement stackElement;
    stackElement.nP = qtreeheaders[fileDesc].root;
    stackElement.region = range;
    openscans[*scanId].searchTree.push(stackElement);
};
```

getNextFoundPoint

The function `getNextFoundPoint` returns the next found point of a valid `OpenScan`. The input arguments of this function are: (i) `scanId` of the corresponding scan and (ii) `foundPoint`, a pointer to a point, which is used to return the found point. The return type of the function is `bool`. It returns 1 if a point is found, otherwise it returns 0 if the scan has finished and no more points have been found. The search is implemented as a depth-first search using the stack `searchTree`, meaning that the child nodes are checked before the neighbor nodes.
The function starts the search by accessing the topmost `SearchStackElement` of `searchTree`. Before checking the node of this stack element, it is checked if the region of the four children overlap the search region. If it does, the corresponding child is inserted into the `searchTree`, since this node has to be checked later. The search region of this child node is set by the coordinates of the parent and the corresponding `x` and `y` of `region`. After all four children have been processed, the point of the node is checked. If the point lays in `scanRange` it is assigned to `foundPoint` and 1 is returned. If the node does not lay in `scanRange`, the next topmost `SearchStackElement` is processed the same way until a point is found. If at the end `searchTree` is empty, the `OpenScan` is closed and 0 is returned to indicate that no point has been found.

```
bool QTree::getNextFoundPoint(int scanId, Point* foundPoint){
    while (!openscans[scanId].searchTree.empty()) {
        //Start with the top of the search-tree-stack
        SearchStackElement stackElem = openscans[scanId].searchTree.top
            ();
        openscans[scanId].searchTree.pop();
        openscans[scanId].visitedNodes++;
        QTreeNode node = getNode(openscans[scanId].fileDesc, stackElem.
            nP);
```

```
        //Push successors to stack if it's no null-pointer
        //and region of successor overlaps the search-range
        if (node.successors[0].blockId != -1 && overlapping(openscans[
            scanId].scanRange,stackElem.region.NE.y, node.coordinates.y,
            stackElem.region.NE.x, node.coordinates.x)) {
            SearchStackElement s;
            s.nP = node.successors[0];
            s.region = Rectangle(stackElem.region.NE.y, node.coordinates
                .y, stackElem.region.NE.x, node.coordinates.x);
            openscans[scanId].searchTree.push(s);
        }
        if (node.successors[1].blockId != -1 && overlapping(openscans[
            scanDesc].scanRange,stackElem.region.NE.y, node.coordinates.y
            , node.coordinates.x, stackElem.region.SW.x)) {
            SearchStackElement s;
            s.nP = node.successors[1];
            s.region = Rectangle(stackElem.region.NE.y, node.coordinates
                .y, node.coordinates.x, stackElem.region.SW.x);
            openscans[scanId].searchTree.push(s);
        }
        if (node.successors[2].blockId != -1 && overlapping(openscans[
            scanDesc].scanRange,node.coordinates.y, stackElem.region.SW.y
            , node.coordinates.x, stackElem.region.SW.x)) {
            SearchStackElement s;
            s.nP = node.successors[2];
            s.region = Rectangle(node.coordinates.y, stackElem.region.SW
                .y, node.coordinates.x, stackElem.region.SW.x);
            openscans[scanId].searchTree.push(s);
        }
        if (node.successors[3].blockId != -1 && overlapping(openscans[
            scanDesc].scanRange, node.coordinates.y, stackElem.region.SW.
            y, stackElem.region.NE.x, node.coordinates.x)) {
            SearchStackElement s;
            s.nP = node.successors[3];
            s.region = Rectangle(node.coordinates.y, stackElem.region.SW
                .y, stackElem.region.NE.x, node.coordinates.x);
            openscans[scanId].searchTree.push(s);
        }
        //Check if the node itself lays in the search-range
        if (inRegion(node.coordinates, openscans[scanDesc].scanRange)) {
            *foundPoint = node.coordinates;
            openscans[scanId].foundPoints++;
            return true;
        }
    }
    closeRegionScan(scanId);
    return false;
};
```

closeRegionScan

This function closes a valid `OpenScan`. It has one input argument `scanId`. First, the function sets the corresponding scan to invalid. Then an empty stack is exchanged with `searchTree`, so the new stack is empty.

```
void QTree::closeRegionScan(int scanId){
    //Set scan to false and empty stack
    openscans[scanId].valid = false;
    stack<SearchStackElement> emptyStack;
    swap(openscans[scanDesc].searchTree, emptyStack);
};
```

4 Experiments

In this chapter the experiments on the implementation of a quadtree on top of a buffer manager are described. These experiments illustrate different good and bad scenarios of the implementation.
The system used for the experiments was a MacBook Air (2012), equipped with a 1.7 GHz Intel Core i5, 4 GB of RAM and a SSD. The operating system was OS X Yosemite.
The experiments are divided into two parts. The first part analyzes the impact of the size of the input data and of the insertion method used (`insertEntry` / `smartInsertEntry`) on the efficiency of insertion and search. The second part focuses on the impact of the parameters of the buffer manager, namely the size of blocks and the buffer size.

4.1 Experiments focusing on the quadtree

The first part of the experiments was done to analyze the efficiency of a quadtree relative to the size of the input data. For this (i) input data of various sizes and (ii) the two insertion methods were used. Insertion and region search on the different quadtrees were tested for five different input sizes from 1 mil. points to 30 mil. points. For each input size four input files were generated with random points ranging from 0 to 1. These uniformly distributed random points were generated with the library `<random>`. For testing the region search, a file with 25 randomly located rectangles for three different edge sizes was generated.
For every input file two quadtrees were created to test the insertion, one using `insertEntry` (from now called naive insertion) and one using `smartInsertEntry` (from now called smart insertion). For each data size two quadtrees were selected to test the region search, one created with the naive insertion and one with the smart insertion. The 75 rectangles were used to perform the region searches on the selected quadtrees. Table 4.1 shows the constellation of all these experiments.
For all these experiments the size of a block was fixed at 10 KB (holding 200 nodes) and the number of pages of the buffer manager was fixed at 3200, meaning the size of the buffer to be 32 MB.
For every generated quadtree the following data was collected: (i) average path length, (ii) maximum path length, (iii) standard deviation of the path length, (iv) requested block reads, (v) effective block reads, (vi) requested block writes, (vii) effective block writes, (vii) time to insert all nodes. The maximum path length is equal to the height of a tree, and the average path length equal to the average depth of all nodes.
For each search the following data was collected: (i) number of visited nodes, (ii) number of found nodes, (iii) requested block reads, (iv) effective block reads, (v) time for finding all nodes.

input size	insertion type (each 4x)	edge size (each 25x)
1 mil.	naive / smart	0,125 / 0,25 / 0,5
5 mil.	naive / smart	0,125 / 0,25 / 0,5
10 mil.	naive / smart	0,125 / 0,25 / 0,5
20 mil.	naive / smart	0,125 / 0,25 / 0,5
30 mil.	naive / smart	0,125 / 0,25 / 0,5

Table 4.1: Experiments with different input size

4.2 Experiments focusing on the buffer manager

The second part of the experiments was done to analyze the impact of the parameters of the buffer manager (block size, number of pages). Keeping the size of the tree fixed at 5 mil. points, the block size and also the number of pages were varied. There were two parts; (i) in the first part the number of pages was fixed while the block size was varied (increasing the block size increases the buffer size) and (ii) in the second part the total buffer size was fixed and both the block size and the number of pages was varied (increasing the block size decreases the number of pages). For both parts the different block sizes were: (i) 2,5 KB, (ii) 5 KB, (iii) 10 KB, (iv) 25 KB, (v) 50 KB. For the experiments with the same number of pages the number of pages was fixed at 3200 (Table 4.2). For the experiments with the same buffer size the size of the buffer was fixed at 32 MB and therefore the number of pages were: (i) 640, (ii) 1280, (iii) 3200, (iv) 6400, (v) 12800 (Table 4.3). The insertion was only done for the smart insertion type. Again for every constellation of the parameters four quadtree were generated and again 25 searches were performed per edge size and for each experiment the same variables were gathered.

input size	block size	# of pages	insertion type (each 4x)	edge size (each 25x)
5 mil.	2,5KB	3200	smart	0,125 / 0,25 / 0,5
5 mil.	5KB	3200	smart	0,125 / 0,25 / 0,5
5 mil.	10KB	3200	smart	0,125 / 0,25 / 0,5
5 mil.	25KB	3200	smart	0,125 / 0,25 / 0,5
5 mil.	50KB	3200	smart	0,125 / 0,25 / 0,5

Table 4.2: Experiments with fixed number of pages

input size	block size	# of pages	insertion type (each 4x)	edge size (each 25x)
5 mil.	2,5KB	12800	smart	0,125 / 0,25 / 0,5
5 mil.	5KB	6400	smart	0,125 / 0,25 / 0,5
5 mil.	10KB	3200	smart	0,125 / 0,25 / 0,5
5 mil.	25KB	1280	smart	0,125 / 0,25 / 0,5
5 mil.	50KB	640	smart	0,125 / 0,25 / 0,5

Table 4.3: Experiments with fixed buffer size

5 Results

5.1 Results of quadtree experiments

The insertion of nodes to a quadtree and the region search have been tested with different tree sizes. Also the quadtrees generated by the naive and the smart insertion method have been differentiated for the results. Figure 5.1 shows the results of the insertion and Figure 5.3 the results of the search.

5.1.1 Insertion

Different quadtrees have been generated with different tree sizes using both the naive and the smart insertion. Figure 5.1(a) shows the average and the maximum path length of the quadtrees relative to the data size (number of nodes). Also the average and the maximum path length of a perfect tree of the same size is plotted. A perfect quadtree is a tree in which all nodes have four child nodes and all leaves have the same depth. The maximum path length of a perfect tree represents also the minimum height of a quadtree for a given number of nodes. The maximum path length of a perfect quadtree can be calculated with following formula, where `N` is the number of nodes and `h` is the height of the tree:

$$
\begin{aligned}
N &= 1 + 4 + 16 + ...4^h \\
N &= \sum_{t=0}^{h} 4^t \\
N &= \frac{4^{h+1} - 1}{3} \\
h &= \log_4(3N + 1) - 1
\end{aligned}
$$

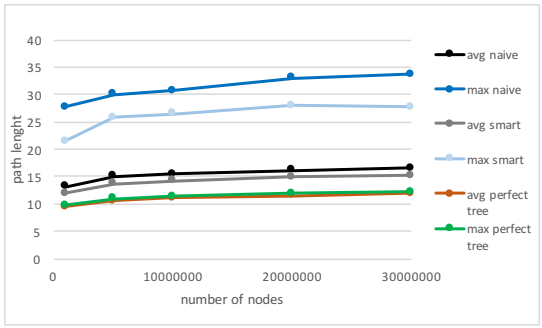
The average path length of a perfect quadtree can be calculated with following formula,

using the before calculated height:

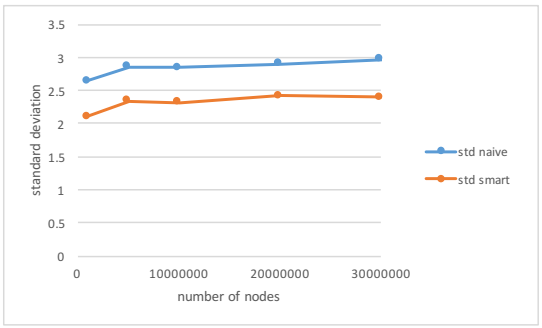
$$
\begin{aligned}
avg_h &= \frac{0 * 1 + 4 * 1 + 16 * 2 + ...4^h * h}{N} \\
avg_h &= \frac{\sum_{t=0}^{h} 4^t t}{N} \\
avg_h &= \frac{\frac{h4^{h+2} - (h+1)4^{h+1} + 4}{(1-4)^2}}{N} \\
avg_h &= \frac{4}{9N}(3 * 4^h h - 4^h + 1)
\end{aligned}
$$

In Figure 5.1(a) it can be seen, that both the maximum and the average path length is lower when the smart insertion is used. For bigger tree sizes the path length increases in a logarithmic way for both methods and also for a perfect quadtree. The difference of average and maximum path length is almost non-existent for the perfect tree whereas it is much bigger for the naive and the smart insertion. Because the naive insertion does not balance at all and therefore has a higher maximum path length, the standard deviation is bigger than for the smart insertion, which can also be verified with Figure 5.1(b).
Figure 5.1(c) shows the average requested reads and the average effective reads for both insertion types for one single inserted point. Because the smart insertion additionally balances leaves of the trees, more blocks are requested. The effective reads are much lower than the requested and they are nearly the same for the two insertion types. Regarding the balancing, the smart insertion type requests blocks which have just been requested before, so the buffer manager does not perform effective reads again. The requested and also the effective reads for both insertion types increase logarithmically with the tree size. The reason is, that the number of reads depends on the average path length, since for every insertion the parent of the new node has to be found by traversing the path from the root to the new parent node. And since the average path length increases logarithmically, the number of reads also increases logarithmically.
Figure 5.1(d) shows the same as Figure 5.1(c) but for the number of writes. It shows that the requested writes are constant. For the naive insertion two writes are requested; one to insert the new node and one to modify the parent node. For the smart insertion it is higher, at about 3.25, as the balancing requires additional write requests. The number of effective writes are not constant, they increase logarithmically and the ones for the smart insertion are slightly higher.
Figure 5.1(e) shows the total time required for inserting all points. It can be seen, that it is increasing at a disproportionate rate. The reason for this can be found in Figure 5.1(f), which shows the time for the insertion of one node. It increases logarithmically basically because the effective reads and writes for one inserted node increase logarithmically. The total time then can be calculated by multiplying the time per insertion with the number of nodes, which increases at a disproportionate rate. The smart insertion requires more time, because of higher effective writes and the additional computation, while effective reads are the same as in naive insertion.

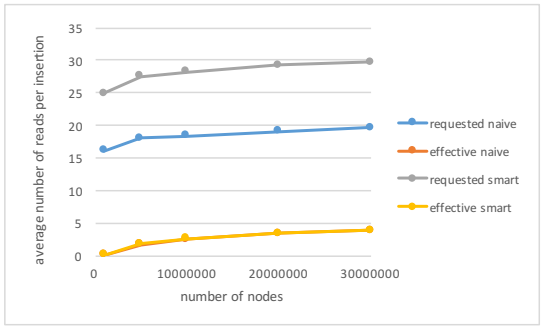
It has to be mentioned that since the inserted points are uniformly distributed the resulting trees are not very unbalanced. In terms of tree heights both the naive and the smart insertion have worse results only by a factor 1,5 than the perfect tree. These experiments represent the average case. But in the worst case scenario, the height of a tree can be much worse. If sorted input data is used, the resulting trees for the naive insertion will have a height of $N-1$ and for the smart insertion a height of $\frac{N-1}{2}$. Figure 5.2 shows an example of the resulting trees, on the left with naive insertion and on the right with smart insertion. The worst case scenario has not been included to the experiments since it takes too much time to generate the quadtree files. Inserting 25'000 sorted points takes as much time as inserting 1 mil. uniformly distributed points. Inserting only one input file with 30 mil sorted points is expected to take several weeks.



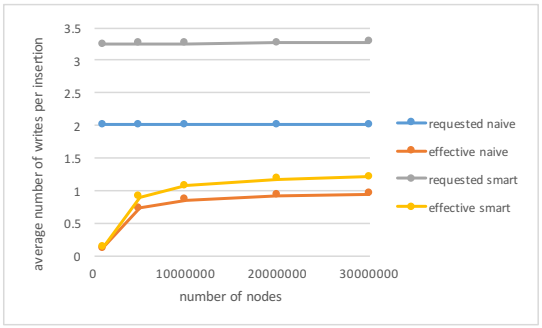
(a) path length



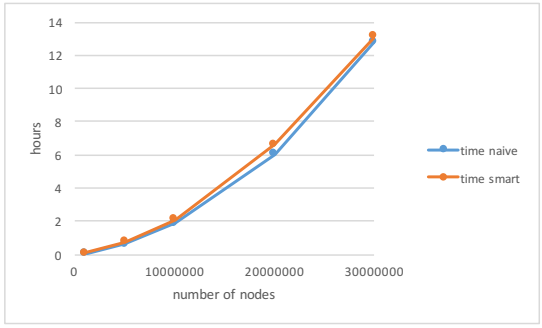
(b) standard deviation of path length



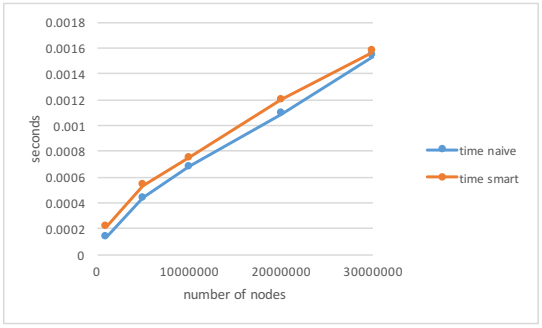
(c) reads



(d) writes



(e) total time



(f) average time

Figure 5.1: Insertion - different input sizes

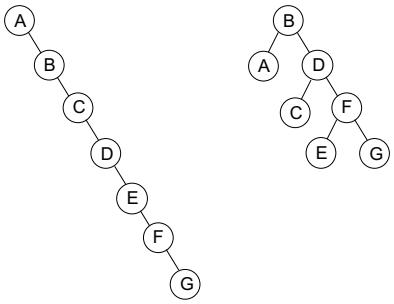


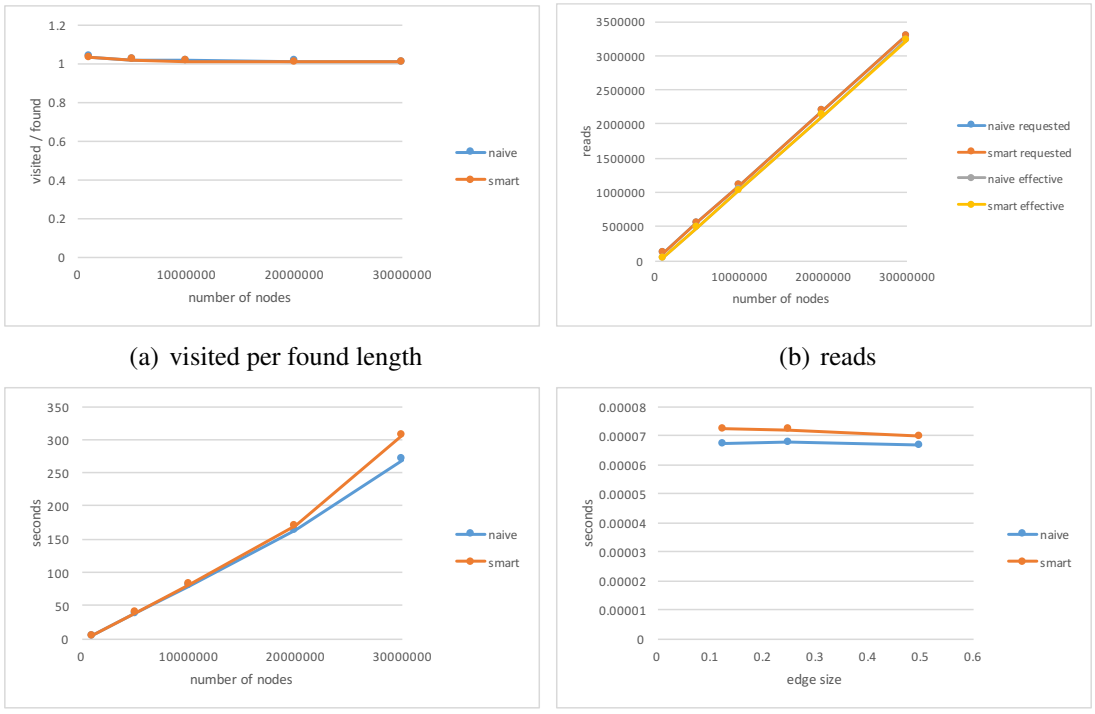
Figure 5.2: Quadtrees generated with sorted points

5.1.2 Search

Various region searches have been performed on quadtrees with different tree sizes. Figure 5.3 shows the results of these experiments. Figure 5.3(a) shows the ratio visited nodes per found nodes. It can be seen, that the ratio is just above one for any tree size. So the number of visited nodes, that are not in the search region, is quite small proving the efficiency of the quadtree. There is almost no difference between this ratio of the naive and smart insertion type.
In Figure 5.3(b) the average requested and the effective reads are plotted. The requested reads are growing linearly with the tree size, because the number of found nodes is also growing with the trees size. Interestingly the effective reads are almost as big as the requested. The reason for this might be found in the implemented way of the region search; it traverses the quadtree like a depth-first search meaning it checks the child nodes before the neighbor nodes. It seems to be more likely that a block contains mainly neighbor nodes. This makes it unlikely that a block is already in the buffer and the effective reads are almost as high as the requested. And there is practically no difference between naive and smart type, as the average path length does not differ much.
Figure 5.3(c) shows the average time for the performed region searches. It grows linearly like the reads. The naive and the smart require almost the same time, strangely enough the smart type is slightly, but not significantly higher. The time for the smart type for 30 mil. tree size deviates from the rest. It seems to be just an outlier, else it can not be explained.
In Figure 5.3(d) the time per found node is plotted for the three edge sizes. It can be seen, that the edge size does not affect the time per found node much. This means it would not make a big difference in time if a region search with a big region is performed or several searches with smaller regions are performed, if the number of found nodes would be the same. Again the smart type requires more time than the naive one, even though it is not a significant variation.

5.2 Results of buffer manager experiments

In the second part of the experiments insertion and the region search have been tested with different block sizes and different number of pages. First, the block size has been increased while holding the number of pages fixed. Second the block size has been increased while the number of pages decreased so that the total buffer size is fixed. For these experiments only



(a) visited per found length

(b) reads

(c) time

(d) time per found node

Figure 5.3: Search - different input sizes

the smart insertion has been used. Figure 5.4 shows the results of the insertion and Figure 5.5 the results of the region search.

5.2.1 Insertion

The same input files have been used to generate quadtrees with different block sizes and number of pages. Because only the parameters of the buffer manager vary, the resulting path length of these quadtrees and the requested reads and writes remain the same. But the effective reads and writes and therefore the time varies. The results are shown in Figure 5.4.

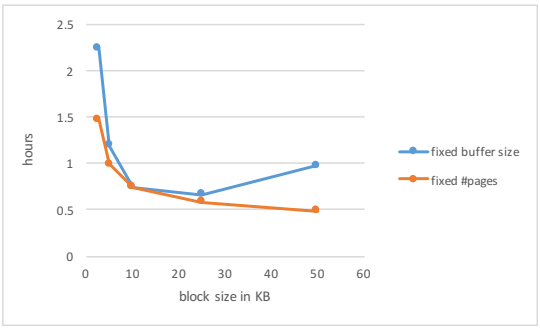
Fixed number of pages

The orange line in Figure 5.4(a) indicates the insertion time for keeping the number of pages fixed (at 3200 pages). This means, that the size of the buffer increases w.r.t. the block size. For the same number of pages it can be seen, that the time is decreasing like a exponential function with negative exponents. This means, that by increasing the size of the buffer (and holding the number of pages fixed), the required time decreases. But beyond a certain level there is only a marginal time difference. Figure 5.4(b) and 5.4(c) show that the effective reads and writes are decreasing stronger than the time. The reason, why the time is not decreasing

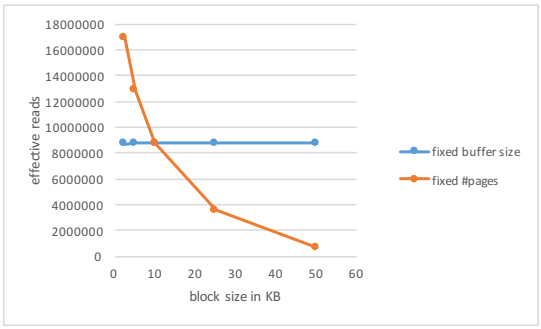
as strong as the reads and writes, is that for a bigger block size the time for each effective read/write increases also linearly.

Fixed size of buffer

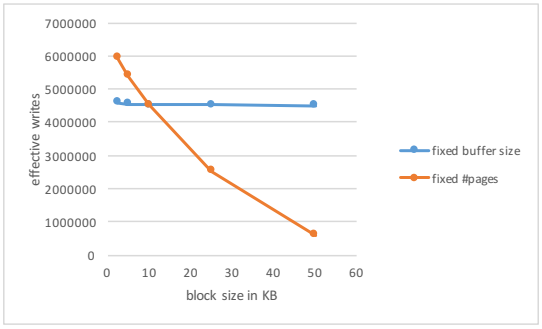
The blue line indicates in Figure 5.4(a) the insertion time for keeping the size of the buffer fixed (at 32MB). This means, that the number of pages decreases w.r.t. the block size. For small block sizes the time is high. While the block size increases the time decreases, but at a certain point the time is slowly increasing again. So for a given size of the buffer, the constellation of the block size and number of pages has a strong impact on the time. Both small blocks plus many pages and big blocks plus less pages are inefficient. For the performance experiments the optimum block size lies between 10KB and 25KB.
Figure 5.4(b) and 5.4(c) show that the effective reads and writes are more or less constant because the number of nodes stored in the buffer stays the same for any size of the blocks. That is why on the one hand the time increases for bigger block sizes, since every read/write needs more time. On the other hand, too many small blocks lead to a higher required time because the linear search in `pages` takes more time. The impact of too many pages can be seen for the block size of 5KB. For a buffer size of 32MB and 12800 pages the time is much higher than for 3200 pages and a buffer size of 8MB. With less pages the time is lower. This trade-off leads to the existence of an optimum block size and number of pages for a given buffer size.



(a) time



(b) effective reads



(c) effective writes

Figure 5.4: Insertion - different block sizes

5.2.2 Search

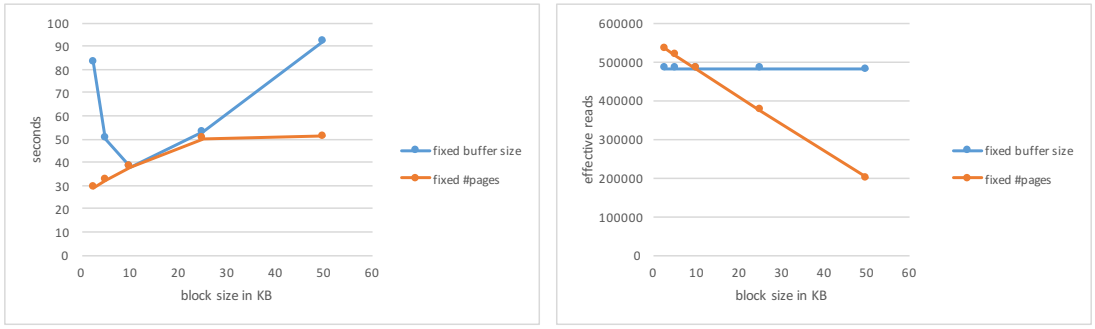
The generated quadtree files have been used to perform region searches to see the impact of the block size and the number of pages. The number of visited/found nodes and the number of read request remains the same as only the parameters of the buffer manager are varied. Only the effective reads and the time varies. The results of the experiments are illustrated in 5.5.

Fixed number of pages

In Figure 5.5(a) it can be seen, that for a fixed number of pages the time for the search increases with the block size in a logarithmic way. At a first glance, it might seem strange, since the buffer size is also increasing w.r.t. the block size. The reason for this can be seen in 5.5(b). The effective reads are decreasing w.r.t. the block size. But doubling the block size does not result in halfing the effective reads. But a double block size means double time to read one block. The lower effective reads can not compensate the additional time of accessing bigger blocks. Thats why the time increases w.r.t the block size. This means, for the region search a big buffer size with big block sizes is inefficient, because the number of effective reads is too high.

Fixed size of buffer

In Figure 5.5(a) the time for different block sizes but fixed buffer size can be seen. For small blocks it is high but decreases w.r.t. the block size. But then at a certain point the time increases. The optimal block size seems to be around 10KB. Again there is a trade-off which lead to this circumstance. On the one hand, the number of pages decreases with the block size. Each time a block is requested, the buffer manager performs a linear search in `pages`. Therefore, more pages lead to more needed time. On the other hand, the number of effective reads is almost constant as it can be seen in Figure 5.5(b). Therefore, for a bigger block size the reading time is increasing because each effective read requires more time.



(a) time (b) effective reads

Figure 5.5: Search - different block sizes

6 Summary and Conclusion

The goal of this thesis was to implement a disk-resident quadtree and evaluate its efficiency. First, I researched on related work to understand the concepts of quadtrees and paging. Then a general purpose buffer manager has been implemented which performs the actual paging so even very big files can be handled. On top of this buffer manager the quadtree has been implemented to store the full tree structure in a file. In a next step the efficiency of this implementation has been evaluated with different experiments.

The experiments have shown that the quadtree performs a region search very efficient, as the number of visited nodes is not much higher than the number of found nodes. Even for big number of nodes the region search is very efficient. The number of effective reads are quite high, paging did not really improve the performance of region search. The time to insert one point increases logarithmically w.r.t. the input size. Therefore inserting all nodes takes linearithmic time (O(n log n)). For insertion, the buffer manager improved the performance by significantly reducing the number of effective reads and writes. The smart insertion method hasn't proved to be more efficient than the naive one for uniformly distributed random points.

Also the experiments have shown that the block size and the number of pages have a strong impact on the efficiency of the quadtree. For a given buffer size there is an optimal value for the block size and the number of pages. These values are depending on the read and write rates on the disk and on the power of the CPU. A bigger size of the buffer with optimal values for the block size and the number of pages could lead to better performance. But beyond a certain level there is only a marginal difference. Since the effective reads are too high, a bigger buffer size due to bigger a block size does not lead to better performance of the region search.

As a future work, it would be interesting to implement the region search, which traverses the quadtree like a breadth-first search (checking the neighbors before the children). If in this way the number of effective reads are decreased, then the region search could be more efficient. Another interesting approach to increase the performance would be to use a binary tree or a hash table to find a block in the pages. The fileId and the blockId can be used as key. An additional priority queue can be used to find the page with the lowest LRU counter. In this way a linear search in `pages` can be avoided. With experiments the advantages and disadvantages of this implementation could be detected.

Bibliography

[EN10] Ramez Elmasri and Sham Navathe. *Fundamentals of database systems*. Pearson Education India, 2010.

[FB74] Raphael A. Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[GTM07] Michael Goodrich, Roberto Tamassia, and David Mount. *DATA STRUCTURES AND ALOGORITHMS IN C++*. John Wiley & Sons, 2007.

[Sam84] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.

[Shn81] Michael Shneier. Two hierarchical linear feature representations: edge pyramids and edge quadtrees. *Computer Graphics and Image Processing*, 17(3):211–224, 1981.

[SW84] Hanan Samet and Robert E Webber. On encoding boundaries with quadtrees. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 16(3):365–369, 1984.