M-Tree Algebra

Contents

1 Overview

This algebra provides the `mtree` type constructor and the following operators: (`DATA` must be of the same type as the attributes indized in the m-tree):

- `_ createmtree [_]`
  Creates a new mtree from a relation or tuple stream.

  Signature: `relation/tuple-stream x <attr-name> -> mtree`
  Example: `pictures createmtree[Pic]`

- `_ createmtree2 [_, _]`
  Like createmtree, but additionally allows to select a mtree-config object.

  Signature: `relation/tuple-stream x <attr-name> x <config-name> -> mtree`
  Example: `pictures createmtree2[Pic, limit80e]`

- `_ createmtree3 [_, _, _]`
  Like createmtree2, but additionally allows to select the distance function and distdata type.

  Signature: `relation/tuple-stream x <attr-name> x <config-name> x <distfun-name> x <distdata-name> -> mtree`
  Example: `pictures createmtree3[Pic, limit80e, euclid, lab256]`

- `_ _ rangesearch [_, _]`
  Returns all tuples of the relation, for which the indized entries lies inside the query range around the query attribute. The relation should be the same that had been used to create the tree.

  Signature: `mtree relation x DATA x real -> tuple stream`
  Example: `pictree pictures rangesearch[pic, 0.2] count`

- `_ _ nnsearch [_, _]`
  Returns all tuples of the relation, which for which the indized entries are the n nearest neigthbours of the query attribute. The relation should be the same that had been used to create the tree.

  Signature: `mtree relation x DATA x int -> tuple stream`
  Example: `pictree pictures nnsearch[p1, 5] count`

2 Headerfile `MTreeAlgebra.h`

January-May 2008, Mirko Dibbert

2.1 Overview

This file contains some defines and constants, which could be used to configurate this algebra.

```
#ifndef __MTREE_ALGEBRA_H__
#define __MTREE_ALGEBRA_H__

#include "GeneralTreeAlgebra.h"

using namespace gta;
using gtree::NodeConfig;
using gtree::NodeTypeId;



//////////////////////////////////////////////////////////////////////
// enables debugging mode for the mtree-algebra:
//////////////////////////////////////////////////////////////////////
//#define __MTREE_DEBUG


//////////////////////////////////////////////////////////////////////
// enables print of statistic infos in the insert method:
```

```
////////////////////////////////////////////////////////////////////
#define __MTREE_PRINT_INSERT_INFO


////////////////////////////////////////////////////////////////////
// enables print of mtree statistics in the out function:
////////////////////////////////////////////////////////////////////
#define __MTREE_OUTFUN_PRINT_STATISTICS


////////////////////////////////////////////////////////////////////
// enables print of count of objects in leaf/right node after split:
////////////////////////////////////////////////////////////////////
// #define MTREE_PRINT_SPLIT_INFO


////////////////////////////////////////////////////////////////////
// enables print of statistic infos in the search methods:
////////////////////////////////////////////////////////////////////
// #define __MTREE_PRINT_SEARCH_INFO


////////////////////////////////////////////////////////////////////
// enables print of statistic infos in the search methods
// to file "mtree.log"
////////////////////////////////////////////////////////////////////
// #define __MTREE_PRINT_STATS_TO_FILE



namespace mtreeAlgebra
{

enum PROMOTE
{ RANDOM, m_RAD, mM_RAD, M_LB_DIST };
```

Enumeration of the implemented promote functions:

- RANDOM : Promotes two random entries.
- m_RAD : Promotes the entries which minimizes the sum of both covering radii.
- mM_RAD**:** Promotes the entries which minimizes the maximum of both covering radii.
- M_LB_DIST : Promotes as first entry the previously promoted element, which is equal to the parent entry. As second entry, the one with maximum distance to the parent entry would be promoted.

```
enum PARTITION
{ GENERALIZED_HYPERPLANE, BALANCED };
```

Enumeration of the implemented partition functions.
Let p_1, p_2 be the promoted items and N_1, N_2 be the nodes containing p_1 and p_2:

- GENERALIZED_HYPERPLANE The algorithm assign an entry e as follows: if $d(e, p_1) \leq d(e, p_2)$, e is assigned to N_1, otherwhise it is assigned to N_2.
- BALANCED : This algorithm alternately assigns the nearest neighbour of p_1 and p_2, which has not yet been assigned, to N_1 and N_2, respectively.

```
//////////////////////////////////////////////////////////////////
// en-/disable caching for all node types
//////////////////////////////////////////////////////////////////
const bool nodeCacheEnabled = true;


//////////////////////////////////////////////////////////////////
// intevall of printing statistic infos in the insert method
// (only used, if __MTREE_PRINT_INSERT_INFO has been defined)
//////////////////////////////////////////////////////////////////
const int insertInfoInterval = 100;
```

The following constants are only default values for the mtree-config objects, that could be changed in some configurations. See the initialize method of the MTreeConfigReg class for details.

```
//////////////////////////////////////////////////////////////////
// default split policy
//////////////////////////////////////////////////////////////////
const PROMOTE defaultPromoteFun = M_LB_DIST;
const PARTITION defaultPartitionFun = BALANCED;


//////////////////////////////////////////////////////////////////
// en-/disable caching seperately for each node type
//////////////////////////////////////////////////////////////////
const bool leafCacheable     = true;
const bool internalCacheable = true;


//////////////////////////////////////////////////////////////////
// max. count of pages for leaf / internal nodes
//////////////////////////////////////////////////////////////////
const unsigned maxLeafPages   = 1;
const unsigned maxIntPages    = 1;


//////////////////////////////////////////////////////////////////
// min. count of entries for leaf / internal nodes
//////////////////////////////////////////////////////////////////
const unsigned minLeafEntries = 3;
const unsigned minIntEntries  = 3;


//////////////////////////////////////////////////////////////////
// max. count of entries for leaf / internal nodes
//////////////////////////////////////////////////////////////////
const unsigned maxLeafEntries = numeric_limits<unsigned>::max();
```

```
const unsigned maxIntEntries = numeric_limits<unsigned>::max();


////////////////////////////////////////////////////////////////////
// priorities of the node types
// (higher priorities result into a higher probablility for
// nodes of the respective type to remain in the node cache)
////////////////////////////////////////////////////////////////////
const unsigned leafPrio     = 0; // default = 0
const unsigned internalPrio = 1; // default = 1


////////////////////////////////////////////////////////////////////
// constants for the node type id's
////////////////////////////////////////////////////////////////////
const NodeTypeId LEAF = 0;
const NodeTypeId INTERNAL = 1;



// define __MTREE_ANALYSE_STATS if __MTREE_PRINT_SEARCH_INFO or
// __MTREE_PRINT_STATS_TO_FILE has been defined
#ifdef __MTREE_PRINT_SEARCH_INFO
  #define __MTREE_ANALYSE_STATS
#else
  #ifdef __MTREE_PRINT_STATS_TO_FILE
  #define __MTREE_ANALYSE_STATS
  #endif
#endif

} // namespace mtreeAlgebra
#endif // #ifndef __MTREE_ALGEBRA_H__
```

3 Headerfile `MTree.h`

January-May 2008, Mirko Dibbert

3.1 Overview

This file contains the `MTree` class and some auxiliary structures.

3.2 Includes and defines

```
#ifndef __MTREE_H__
#define __MTREE_H__

#include "MTreeBase.h"
#include "MTreeSplitpol.h"
#include "MTreeConfig.h"
#include "SecondoInterface.h"
#include "AlgebraManager.h"

extern SecondoInterface* si;
extern AlgebraManager* am;

namespace mtreeAlgebra
{
```

3.3 Struct `SearchBestPathEntry`:

This struct is needed in the `insert` method of `mtree`.

```
struct SearchBestPathEntry
{
    SearchBestPathEntry(
            InternalEntry* _entry, double _dist,
            unsigned _index) :
        entry(_entry), dist(_dist), index(_index)
    {}

    mtreeAlgebra::InternalEntry* entry;
    double dist;
    unsigned index;
};
```

3.3.1 Struct `SearchBestPathEntry`:

This struct is needed in the `mtree::rangeSearch` method.

```
struct RemainingNodesEntry
{
    SmiRecordId nodeId;
    double dist;

    RemainingNodesEntry(
            SmiRecordId _nodeId, double _dist) :
        nodeId(_nodeId), dist (_dist)
    {}
};
```

3.3.2 Struct `SearchBestPathEntry`:

This struct is needed in the `mtree::nnSearch` method.

```
struct RemainingNodesEntryNNS
{
    SmiRecordId nodeId;
    double minDist;
    double distQueryParent;

    RemainingNodesEntryNNS(
            SmiRecordId _nodeId, double _distQueryParent,
            double _minDist) :
        nodeId(_nodeId), minDist(_minDist),
        distQueryParent(_distQueryParent)
    {}

    bool operator > (const RemainingNodesEntryNNS& op2) const
    { return (minDist > op2.minDist); }
};
```

3.3.3 Struct `NNEntry`:

This struct is needed in the `mtree::nnSearch` method.

```
struct NNEntry
{
    TupleId tid;
    double dist;

    NNEntry(TupleId _tid, double _dist)
    : tid(_tid), dist(_dist)
    {}

    bool operator < (const NNEntry& op2) const
    {
        if (((tid == 0) && (op2.tid == 0)) ||
            ((tid != 0) && (op2.tid != 0)))
        {
            return (dist < op2.dist);
        }
        else  if ((tid == 0) && (op2.tid != 0))
        {
            return true;
        }
        else // ((tid != 0) && (op2.tid == 0))
        {
            return false;
        }
    }
};
```

3.4 Struct `Header`

```
struct Header
    : public gtree::Header
{
    Header()
        : gtree::Header(), initialized(false)
    {
        distfunName[0] = '\0';
        configName[0] = '\0';
    }

    STRING_T distfunName; // name of the used metric
    STRING_T configName;  // name of the MTreeConfig object
    DistDataId dataId;    // id of the used distdata type
    bool initialized;     // true, if the mtree has been initialized
};
```

3.5 Class `MTree`

```
class MTree
```

```
    : public gtree::Tree<Header>
{

public:
```

Default Constructor, creates a new m-tree.

```
    inline MTree(bool temporary = false)
        : gtree::Tree<Header>(temporary), splitpol(false)
    {}
```

Constructor, opens an existing tree.

```
    inline MTree(const SmiFileId fileId)
        : gtree::Tree<Header>(fileId), splitpol(false)
    {
        if (header.initialized)
        {
            initialize();
            registerNodePrototypes();
        }
    }
```

Default copy constructor

```
    inline MTree(const MTree& mtree)
        : gtree::Tree<Header>(mtree), splitpol(false)
    {
        if (mtree.isInitialized())
            initialize();
    }
```

Destructor

```
    inline ~MTree()
    {
        if (splitpol)
            delete splitpol;
    }
```

Initializes a new created m-tree. This method must be called, before a new tree could be used.

```
    void initialize(
            DistDataId dataId,
            const string &distfunName,
            const string &configName);
```

Creates a new LeafEntry from `attr` and inserts it into the mtree.

```
    void insert(Attribute *attr, TupleId tupleId);
```

Creates a new LeafEntry from `data` and inserts it into the mtree.

```
void insert(DistData* data, TupleId tupleId);
```

Inserts a new entry into the mtree.

```
void insert(LeafEntry* entry, TupleId tupleId);
```

Returns all entries, wich have a maximum distance of `rad` to the given `Attribute` object in the result list.

```
inline void rangeSearch(
        Attribute *attr, const double &rad,
        list<TupleId> *results)
{ rangeSearch(df_info.getData(attr), rad, results); }
```

Returns all entries, wich have a maximum distance of `rad` to the given `DistData` object in the result list.

```
void rangeSearch(
        DistData *data, const double &rad,
        list<TupleId> *results);
```

Returns the `nncount` nearest neighbours ot the `Attribute` object in the result list.

```
inline void nnSearch(
        Attribute *attr, int nncount, list<TupleId> *results)
{ nnSearch(df_info.getData(attr), nncount, results); }
```

Returns the `nncount` nearest neighbours ot the `DistData` object in the result list.

```
void nnSearch(
        DistData *data, int nncount, list<TupleId> *results);
```

Returns the name of the assigned type constructor.

```
inline string typeName()
{ return df_info.data().typeName(); }
```

Returns the name of the assigned distance function.

```
inline string distfunName()
{ return header.distfunName; }
```

Returns the name of the assigned distdata type.

```
inline string dataName()
{ return df_info.data().name(); }
```

Returns the id of the assigned distdata type.

```
inline DistDataId& dataId()
{ return header.dataId; }
```

Returns the name of the used `MTreeConfig` object.

```
        inline string configName()
        { return header.configName; }
```

Returns true, if the m-tree has already been initialized.

```
        inline bool isInitialized() const
        { return header.initialized; }
```

Prints some infos about the tree to cmsg.info().

```
        void printTreeInfos()
        {
            cmsg.info() << endl
                << "<mtree infos>" << endl
                << "   entries               : "
                << entryCount() << endl
                << "   height                : "
                << height() << endl
                << "   internal nodes        : "
                << internalCount() << endl
                << "   leaf nodes            : "
                << leafCount() << endl
                << "   assigned config       : "
                << configName() << endl
                << "   assigned type         : "
                << typeName() << endl
                << "   assigned distfun      : "
                << header.distfunName << endl
                << "   assigned distdata type : "
                << df_info.data().name() << endl
                << endl;
            cmsg.send();
        }

    private:
        Splitpol* splitpol;  // reference to chosen split policy
        DistfunInfo df_info; // assigned DistfunInfo object
        MTreeConfig config;  // assigned MTreeConfig object
```

Adds prototypes for the avaliable node types.

```
        void registerNodePrototypes();
```

Initializes distfunInfo splitpol objects and calls the `registerNodePrototypes` method. This method needs an initialized header to work.

```
        void initialize();
```

Splits an node by applying the split policy defined in the MTreeConfing object.

```
    void split();
}; // MTree

} // namespace mrteeAlgebra
#endif // ifdef __MTREE_H__
```

4 Headerfile `MTreeBase.h`

January-May 2008, Mirko Dibbert

4.1 Overview

This headerfile implements entries and nodes of the mtree datastructure.

4.2 Includes and defines

```
#ifndef __MTREE_BASE_H__
#define __MTREE_BASE_H__

#include "RelationAlgebra.h"
#include "MTreeAlgebra.h"

namespace mtreeAlgebra
{

using gtree::NodePtr;
```

4.3 Class *LeafEntry*

```
class LeafEntry
        : public gtree::LeafEntry
{
```

```
    friend class InternalEntry;

    public:
```

Default constructor.

```
        inline LeafEntry()
        {}
```

Constructor (creates a new leaf entry with given values).

```
        inline LeafEntry(
                TupleId _tid, DistData *_data, double _dist = 0)
            : m_tid(_tid), m_data(_data), m_dist(_dist)
        {
            #ifdef __MTREE_DEBUG
            assert(m_data);
            #endif
        }
```

Default copy constructor.

```
        inline LeafEntry(const LeafEntry &e)
            : m_tid(e.m_tid), m_data(new DistData(*e.m_data)),
              m_dist(e.m_dist)
        {}
```

Destructor.

```
        inline ~LeafEntry()
        { delete m_data; }
```

Returns the covering raidius of the entry (always 0 for leafes).

```
        inline double rad() const
        { return 0; }
```

Returns the tuple id of the entry.

```
        inline TupleId tid() const
        { return m_tid; }
```

Returns distance of the entry to the parent node.

```
        inline double dist() const
        { return m_dist; }
```

Sets a new distance to parent.

```
        inline void setDist(double dist)
        { m_dist = dist; }
```

Returns a reference to the `DistData` object.

```
inline DistData *data()
{
    #ifdef __MTREE_DEBUG
    assert(m_data);
    #endif

    return m_data;
}
```

Writes the entry to buffer and increases offset (defined inline, since this method is called only once from Node::write).

```
inline void write(char *const buffer, int &offset) const
{
    gtree::LeafEntry::write(buffer, offset);

    // write tuple-id
    memcpy(buffer+offset, &m_tid, sizeof(TupleId));
    offset += sizeof(TupleId);

    // write distance to parent node
    memcpy(buffer+offset, &m_dist, sizeof(double));
    offset += sizeof(double);

    // write m_data object
    m_data->write(buffer, offset);
}
```

Reads the entry from buffer and increases offset (defined inline, since this method is called only once from Node::read).

```
inline void read(const char *const buffer, int &offset)
{
    gtree::LeafEntry::read(buffer, offset);

    // read tuple-id
    memcpy(&m_tid, buffer+offset, sizeof(TupleId));
    offset += sizeof(TupleId);

    // read distance to parent node
    memcpy(&m_dist, buffer+offset, sizeof(double));
    offset += sizeof(double);

    // read m_data object
    m_data = new DistData(buffer, offset);
}
```

Returns the size of the entry on disc.

```
inline size_t size()
```

```
    {
        return gtree::LeafEntry::size() +
            sizeof(TupleId) + // m_tid
            sizeof(double) +  // m_dist
            sizeof(size_t) +  // size of DistData object
            m_data->size();   // m_data of DistData object
    }

private:
    TupleId     m_tid;   // tuple-id of the entry
    DistData    *m_data; // m_data obj. for m_dist. computations
    double m_dist;  // distance to parent node
};
```

4.4 Class *InternalEntry*

```
class InternalEntry
        : public gtree::InternalEntry
{
public:
```

Default constructor (used to read the entry).

```
  inline InternalEntry()
  {}
```

Constructor (creates a new internal entry with given values).

```
    inline InternalEntry(
            const InternalEntry &e, double _rad,
            SmiRecordId _chield)
        : gtree::InternalEntry(_chield),
          m_dist(e.m_dist), m_rad(_rad),
          m_data(new DistData(*e.m_data))
    {}
```

Constructor (creates a new internal entry from a leaf entry).

```
    inline InternalEntry(
            const LeafEntry &e, double _rad,
            SmiRecordId _chield)
        : gtree::InternalEntry(_chield),
          m_dist(e.m_dist), m_rad(_rad),
          m_data(new DistData(*e.m_data))
    {}
```

Destructor.

```
    inline ~InternalEntry()
    { delete m_data; }
```

Returns distance of the entry to the parent node.

```
inline double dist() const
{ return m_dist; }
```

Returns the covering radius of the entry.

```
inline double rad() const
{ return m_rad; }
```

Returns a reference to the `DistData` object.

```
inline DistData *data()
{
    #ifdef __MTREE_DEBUG
    assert(m_data);
    #endif

    return m_data;
}
```

Sets a new distance to parent.

```
inline void setDist(double dist)
{ m_dist = dist; }
```

Sets a new covering radius.

```
inline void setRad(double rad)
{ m_rad = rad; }
```

Writes the entry to buffer and increases offset (defined inline, since this method is called only once from Node::read).

```
inline void write(char *const buffer, int &offset) const
{
    gtree::InternalEntry::write(buffer, offset);

    // write distance to parent node
    memcpy(buffer+offset, &m_dist, sizeof(double));
    offset += sizeof(double);

    // write covering radius
    memcpy(buffer+offset, &m_rad, sizeof(double));
    offset += sizeof(double);

    // write m_data object
    m_data->write(buffer, offset);
}
```

Reads the entry from buffer and increases offset (defined inline, since this method is called only once from Node::read).

```
    void read(const char *const buffer, int &offset)
    {
        gtree::InternalEntry::read(buffer, offset);

        // read distance to parent node
        memcpy(&m_dist, buffer+offset, sizeof(double));
        offset += sizeof(double);

        // read covering radius
        memcpy(&m_rad, buffer+offset, sizeof(double));
        offset += sizeof(double);

        // read m_data object
        m_data = new DistData(buffer, offset);
    }
```

Returns the size of the entry on disc.

```
    inline size_t size()
    {
        return gtree::InternalEntry::size() +
            2*sizeof(double) + // m_dist, m_rad
            sizeof(size_t) +   // size of DistData object
            m_data->size();    // m_data of DistData object
    }

private:
    double m_dist;  // distance to parent node
    double m_rad;   // covering radius
    DistData    *m_data; // m_data obj. for m_dist. computations
};
```

4.5 Typedefs

```
typedef gtree::LeafNode<LeafEntry> LeafNode;
typedef gtree::InternalNode<InternalEntry> InternalNode;

typedef LeafNode* LeafNodePtr;
typedef InternalNode* InternalNodePtr;

} // namespace mtreeAlgebra
#endif // #ifndef __MTREE_BASE_H__
```

5 Headerfile MTreeConfig.h

January-May 2008, Mirko Dibbert

5.1 Overview

This headerfile contains the MTreeConfigReg class, which provides a set of configurations. Each configuration is identified with a unique name and sets the used split policy as well as the min/max count of entries and max count of pages per node.

All avaliable config objects are defined in the initialize function (file MTreeConfig.cpp) and could be selected with the createmtree2 or createmtree3 operator (createmtree uses the default values).

5.2 Includes and defines

```
#ifndef __MTREE_CONFIG_H__
#define __MTREE_CONFIG_H__

#include "MTreeAlgebra.h"

namespace mtreeAlgebra
{

// name of the default config
const string CONFIG_DEFAULT("default");
```

5.3 Struct *MTreeConfig*

```
struct MTreeConfig
{
```

Config objects for all node types.

```
    NodeConfig leafNodeConfig;
    NodeConfig internalNodeConfig;
```

This parameters contain the promote and partition functions, which should be used.

```
    PROMOTE promoteFun;
    PARTITION partitionFun;
```

Constructor (creates object with default values).

```
    MTreeConfig()
         : leafNodeConfig(
                LEAF, leafPrio, minLeafEntries,
                maxLeafEntries, maxLeafPages, leafCacheable),
           internalNodeConfig(
                INTERNAL, internalPrio, minIntEntries,
                maxIntEntries, maxIntPages, internalCacheable),
           promoteFun(defaultPromoteFun),
           partitionFun(defaultPartitionFun)
     {}
```

Constructor (creates objects with the given parameters).

```
     MTreeConfig(
             NodeConfig _leafNodeConfig,
             NodeConfig _internalNodeConfig,
             PROMOTE _promoteFun,
             PARTITION _partitionFun)
         : leafNodeConfig(_leafNodeConfig),
           internalNodeConfig(_internalNodeConfig),
           promoteFun(_promoteFun),
           partitionFun(_partitionFun)
     {}
};  // struct MTreeConfig
```

5.4 Class `MTreeConfigReg`

```
class MTreeConfigReg
{

public:
```

This method returns the specified `MTreeConfig` object. If no such object could be found, the method returns a new object with default values.

```
        static MTreeConfig getConfig(const string &name);
```

Returns true, if the specified `MTreeConfig` object is defiend.

```
        static bool isDefined(const string &name);
```

Returns a string with the names of all defined config objects.

```
        static string definedNames();
```

Registeres all config objects.

```
        static void initialize();

    private:
        static map<string, MTreeConfig> configs;
        static bool initialized;
    }; // class MTreeConfigReg

    } // namespace mtreeAlgebra
    #endif // #ifdef __MTREE_CONFIG_H__
```

6 Headerfile MTreeSplitpol.h

January-May 2008, Mirko Dibbert

6.1 Overview

This headerfile contains all defined promote- and partition-functions. If a node should be splitted, the promote function select two entries, which should be used as routing entries in the parent node. Afterwards, the partition function divides the origin entries to two entry-vectors, whereas each contains one of the promoted entries.

The partition functions must store the promoted elements as first elements in the new entry vectors, which is e.g. needed in MLB_PROM to determine the entry which is used as routing entry in the parent node.

6.2 Includes and defines

```
#ifndef __SPLITPOL_H__
#define __SPLITPOL_H__

#include "MTreeBase.h"

namespace mtreeAlgebra
{
```

This struct is used in Balanced_Part as entry in the entry-list.

```
template<class TEntry>
struct BalancedPromEntry
{
  TEntry* entry;
  double distToL, distToR;

  BalancedPromEntry(
      TEntry* entry_, double distToL_, double distToR_)
  : entry(entry_), distToL(distToL_), distToR(distToR_) {}
};



class Splitpol; // forward declaration
```

Class `GenericSplitpol`:

This template class contains the defined promote- and partitions functions and is desinged as template class to avoid typecasts for every access to the nodes.

```
template<class TNode, class TEntry>
class GenericSplitpol
{
friend class Splitpol;
```

Constructor.

```
  GenericSplitpol(PROMOTE promId, PARTITION partId, Distfun metric);
```

This function applies the split policy, which had been selected in the constructor. After that, the nodes `_lhs` and `_rhs` contain the entries. The promoted entries are stored in the `promL` and `promR` members, which are accessed from the `Splitpol::apply` method after the split.

```
  inline void apply(TNode* _lhs, TNode* _rhs,
                    SmiRecordId lhs_id, SmiRecordId rhs_id,
                    bool _isLeaf)
  {
    isLeaf = _isLeaf;
    lhs = _lhs;
    rhs = _rhs;

    // create empty vector and swap it with current entry vector
    entries = new vector<TEntry*>();
    entries->swap(*lhs->entries());

    entriesL = lhs->entries();
    entriesR = rhs->entries();

    (this->*promFun)();
    (this->*partFun)();

    promL = new InternalEntry(*((*entries)[promLId]), radL, lhs_id);
    promR = new InternalEntry(*((*entries)[promRId]), radR, rhs_id);
```

```
  delete entries;
}

vector<TEntry*>* entries;  // contains the original entry-vector
vector<TEntry*>* entriesL; // new entry vector for left node
vector<TEntry*>* entriesR; // new entry vector for right node

unsigned promLId; // index of the left promoted entry
unsigned promRId; // index of the right promoted entry

InternalEntry* promL; // promoted Entry for left node
InternalEntry* promR; // promoted Entry for right node

double radL, radR; // covering radii of the prom-entries
double* distances; // array of precmputed distances

bool isLeaf; // true, if the splitted node is a leaf node

TNode *lhs, *rhs; // contains the origin and the new node

Distfun metric; // selected metric.
void (GenericSplitpol::*promFun)(); // selected promote function
void (GenericSplitpol::*partFun)(); // selected partition function
```

Promote functions:

The following methods promote two objects in the entries list and store their indizes in `m_promL` and `m_promR`.

```
void Rand_Prom();
```

This method promtes two randomly selected elements.

```
void MRad_Prom();
```

Promotes the entries which minimizes the sum of both covering radii.

```
void MMRad_Prom();
```

Promotes the entries which minimizes the maximum of both covering radii.

```
void MLB_Prom();
```

Promotes as first entry the previously promoted element, which should be equal to the parent entry. As second entry, the one with maximum distance to parent would be promoted.

Partition functions:

The following methods splits the entries in `m_entries` to `m_entriesL` and `m_entriesR`.

```
void Hyperplane_Part();
```

Assign an entry e to the entry vector, which has the nearest distance between e and the respective promoted element.

```
void Balanced_Part();
```

Alternately assigns the nearest neighbour of m_promL and m_promR, which has not yet been assigned, to m_entriesL and m_entriesR, respectively.

```
}; // class Splitpol
```

6.3 Class `Splitpol`

```
class Splitpol
{

public:
```

Constructor.

```
Splitpol(PROMOTE promId, PARTITION partId, Distfun _metric)
: internalSplit(promId, partId, _metric),
  leafSplit(promId, partId, _metric)
{}
```

This function splits the lhs node. rhs should be an empty node of the same type.

```
inline void apply(NodePtr lhs, NodePtr rhs, bool isLeaf)
{
  if (isLeaf)
  {
    leafSplit.apply(lhs->cast<LeafNode>(), rhs->cast<LeafNode>(),
                    lhs->getNodeId(), rhs->getNodeId(), isLeaf);
    promL = leafSplit.promL;
    promR = leafSplit.promR;
  }
  else
  {
    internalSplit.apply(lhs->cast<InternalNode>(), rhs->cast<InternalNode>(),
                    lhs->getNodeId(), rhs->getNodeId(), isLeaf);
    promL = internalSplit.promL;
    promR = internalSplit.promR;
  }
}

inline InternalEntry* getPromL()
{ return promL; }
```

Returns the routing entry for left node (distance to parent and pointer to chield node needs to be set in from the caller).

```
inline InternalEntry* getPromR()
{ return promR; }
```

Returns the routing entry for right node (distance to parent and pointer to chield node needs to be set in from the caller).

```
private:
  GenericSplitpol<InternalNode, InternalEntry> internalSplit;
  GenericSplitpol<LeafNode, LeafEntry> leafSplit;
  InternalEntry* promL; // promoted Entry for left node
  InternalEntry* promR; // promoted Entry for right node

}; // class Splitpol
```

6.4 Implementation part for `GenericSplitpol` methods

GenericSplitpol Constructor:

```
template<class TNode, class TEntry>
GenericSplitpol<TNode, TEntry>::GenericSplitpol(
    PROMOTE promId, PARTITION partId, Distfun _metric)
: distances(0)
{
  metric = _metric;
  srand(time(0)); // needed for Rand_Prom

  // init promote function
  switch (promId)
  {
    case RANDOM:
      promFun = &GenericSplitpol::Rand_Prom;
      break;

    case m_RAD:
      promFun = &GenericSplitpol::MRad_Prom;
      break;

    case mM_RAD:
      promFun = &GenericSplitpol::MMRad_Prom;
      break;

    case M_LB_DIST:
      promFun = &GenericSplitpol::MLB_Prom;
      break;
  }

  // init partition function
  switch (partId)
  {
    case GENERALIZED_HYPERPLANE:
      partFun = &GenericSplitpol::Hyperplane_Part;
      break;

    case BALANCED:
      partFun = &GenericSplitpol::Balanced_Part;
      break;
  }
}
```

Method *RandProm* :

```
template<class TNode, class TEntry>
void GenericSplitpol<TNode, TEntry>::Rand_Prom()
{
  unsigned pos1 = rand() % entries->size();
  unsigned pos2 = rand() % entries->size();
  if (pos1 == pos2)
  {
    if (pos1 == 0)
      pos1++;
    else
      pos1--;
  }

  promLId = pos1;
  promRId = pos2;
}
```

Method *MRad_Prom* :

```
template<class TNode, class TEntry>
void GenericSplitpol<TNode, TEntry>::MRad_Prom()
{
  // precompute distances
  distances = new double[entries->size() * entries->size()];
  for (unsigned i=0; i< entries->size(); i++)
    distances[i*entries->size() + i] = 0;

  for (unsigned i=0; i < (entries->size()-1); i++)
    for (unsigned j=(i+1); j < entries->size(); j++)
    {
      double dist;
      (*metric)((*entries)[i]->data(),
                (*entries)[j]->data(), dist);
      distances[i*entries->size() + j] = dist;
      distances[j*entries->size() + i] = dist;
    }

  bool first = true;
  double minRadSum;
  unsigned bestPromLId = 0;
  unsigned bestPromRId = 1;

  for (unsigned i=0; i < (entries->size()-1); i++)
    for (unsigned j=(i+1); j < entries->size(); j++)
    {
      // call partition function with promoted elements i and j
      promLId = i;
      promRId = j;
      (this->*partFun)();

      if (first)
```

```
      {
        minRadSum = (radL + radR);
        first = false;
      }
      else
      {
        if ((radL + radR) < minRadSum)
        {
          minRadSum = (radL + radR);
          bestPromLId = i;
          bestPromRId = j;
        }
      }
    }

  promLId = bestPromLId;
  promRId = bestPromRId;

  // remove array of precomputed distances
  delete[] distances;
  distances = 0;
}
```

Method *MMRadProm* :

```
template<class TNode, class TEntry>
void GenericSplitpol<TNode, TEntry>::MMRad_Prom()
{
  // precompute distances
  distances = new double[entries->size() * entries->size()];
  for (unsigned i=0; i< entries->size(); i++)
    distances[i*entries->size() + i] = 0;

  for (unsigned i=0; i < (entries->size()-1); i++)
    for (unsigned j=(i+1); j < entries->size(); j++)
    {
      double dist;
      (*metric)((*entries)[i]->data(),
                (*entries)[j]->data(), dist);
      distances[i*entries->size() + j] = dist;
      distances[j*entries->size() + i] = dist;
    }

  bool first = true;
  double minMaxRad;
  unsigned bestPromLId = 0;
  unsigned bestPromRId = 1;

  for (unsigned i=0; i < (entries->size()-1); i++)
    for (unsigned j=(i+1); j < entries->size(); j++)
    {
      // call partition function with promoted elements i and j
      promLId = i;
```

```
        promRId = j;
        (this->*partFun)();

        if (first)
        {
          minMaxRad = max(radL, radR);
          first = false;
        }
        else
        {
          if (max(radL, radR) < minMaxRad)
          {
            minMaxRad = max(radL, radR);
            bestPromLId = i;
            bestPromRId = j;
          }
        }
      }

    promLId = bestPromLId;
    promRId = bestPromRId;

    // remove array of precomputed distances
    delete[] distances;
    distances = 0;
  }
```

Method *MLBProm* :

```
  template<class TNode, class TEntry>
  void GenericSplitpol<TNode, TEntry>::MLB_Prom()
  {
    #ifdef __MTREE_DEBUG
    assert ((*entries)[0]->dist() == 0);
    #endif

    promLId = 0;
    promRId = 1;
    double maxDistToParent = (*entries)[1]->dist();
    for (unsigned i=2; i < entries->size(); i++)
    {
      double dist = (*entries)[i]->dist();
      if (dist > maxDistToParent)
      {
        maxDistToParent = dist;
        promRId = i;
      }
    }
  }
```

Method *HyperplanePart* :

```
  template<class TNode, class TEntry>
```

```
void GenericSplitpol<TNode, TEntry>::Hyperplane_Part()
{
  entriesL->clear();
  entriesR->clear();

  entriesL->push_back((*entries)[promLId]);
  entriesR->push_back((*entries)[promRId]);

  (*entries)[promLId]->setDist(0);
  (*entries)[promRId]->setDist(0);

  radL = (*entries)[promLId]->rad();
  radR = (*entries)[promRId]->rad();

  for (size_t i=0; i < entries->size(); i++)
  {
    if ((i != promLId) && (i != promRId))
    {
      // determine distances to promoted elements
      double distL, distR;
      if (distances)
      {
          unsigned distArrOffset = i * entries->size();
          distL = distances[distArrOffset + promLId];
          distR = distances[distArrOffset + promRId];
      }
      else
      {
        (*metric)(((*entries)[i])->data(),
                    ((*entries)[promLId])->data(), distL);
        (*metric)(((*entries)[i])->data(),
                    ((*entries)[promRId])->data(), distR);
      }

      /* push entry i to list with nearest promoted entry and update
         distance to parent and covering radius */
      if (distL < distR)
      {
        if (isLeaf)
          radL = max(radL, distL);
        else
          radL = max(radL, distL + (*entries)[i]->rad());

        entriesL->push_back((*entries)[i]);
        entriesL->back()->setDist(distL);
      }
      else
      {
        if (isLeaf)
          radR = max(radR, distR);
        else
          radR = max(radR, distR + (*entries)[i]->rad());
```

```
        entriesR->push_back((*entries)[i]);
        entriesR->back()->setDist(distR);
      }
    }
  }
}
```

Method *BalancedPart* :

```
template<class TNode, class TEntry>
void GenericSplitpol<TNode, TEntry>::Balanced_Part()
{
  entriesL->clear();
  entriesR->clear();

  entriesL->push_back((*entries)[promLId]);
  entriesR->push_back((*entries)[promRId]);

  (*entries)[promLId]->setDist(0);
  (*entries)[promRId]->setDist(0);

  radL = (*entries)[promLId]->rad();
  radR = (*entries)[promRId]->rad();

  /* copy entries into entries (the list contains the entries
     together with its distances to the promoted elements */
  list<BalancedPromEntry<TEntry> > entriesCpy;
  for (size_t i=0; i < entries->size(); i++)
  {
    if ((i != promLId) && (i != promRId))
    {
      double distL, distR;
      if (distances)
      {
          unsigned distArrOffset = i * entries->size();
          distL = distances[distArrOffset + promLId];
          distR = distances[distArrOffset + promRId];
      }
      else
      {
        DistData* data = (*entries)[i]->data();
        DistData* dataL = (*entries)[promLId]->data();
        DistData* dataR = (*entries)[promRId]->data();
        (*metric)(data, dataL, distL);
        (*metric)(data, dataR, distR);
      }

      entriesCpy.push_back(
          BalancedPromEntry<TEntry> (((*entries)[i]), distL, distR));
    }
  }

  /* Alternately assign the nearest neighbour of promL resp.
```

```
   promR to entriesL resp. entriesR and remove it from
   entries. */
bool assignLeft = true;
while (!entriesCpy.empty())
{
  if (assignLeft)
  {
    typename list<BalancedPromEntry<TEntry> >::iterator
        nearestPos = entriesCpy.begin();

    typename list<BalancedPromEntry<TEntry> >::iterator
        iter = entriesCpy.begin();

    while (iter  != entriesCpy.end())
    {
      if ((*iter).distToL < (*nearestPos).distToL)
      {
        nearestPos = iter;
      }
      iter++;
    }

    double distL = (*nearestPos).distToL;
    if (isLeaf)
      radL = max(radL, distL);
    else
      radL = max(radL, distL + (*nearestPos).entry->rad());

    entriesL->push_back((*nearestPos).entry);
    entriesL->back()->setDist(distL);
    entriesCpy.erase (nearestPos);
  }
  else
  {
    typename list<BalancedPromEntry<TEntry> >::iterator
        nearestPos = entriesCpy.begin();

    typename list<BalancedPromEntry<TEntry> >::iterator
        iter = entriesCpy.begin();

    while (iter  != entriesCpy.end())
    {
      if ((*iter).distToL < (*nearestPos).distToR)
      {
        nearestPos = iter;
      }
      iter++;
    }
    double distR = (*nearestPos).distToR;
    if (isLeaf)
      radR = max(radR, distR);
    else
      radR = max(radR, distR + (*nearestPos).entry->rad());
```

```
      entriesR->push_back((*nearestPos).entry);
      entriesR->back()->setDist(distR);
      entriesCpy.erase (nearestPos);
    }
      assignLeft = !assignLeft;
  }
}

} // namespace mtreeAlgebra
#endif // #ifndef __SPLITPOL_H__
```