M-Tree Algebra

Contents

1 Headerfile `MTreeAlgebra.h`

January-February 2008, Mirko Dibbert

1.1 Overview

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

1.2 Includes and defines

```
#ifndef __MTREE_ALGEBRA_H
#define __MTREE_ALGEBRA_H

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


////////////////////////////////////////////////////////////////////
// enables debugging mode for general tree algebra framework:
////////////////////////////////////////////////////////////////////
// #define GTAF_DEBUG


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

```
// (should be replaced by progress operator version)
////////////////////////////////////////////////////////////////////
#define MTREE_PRINT_INSERT_INFO

////////////////////////////////////////////////////////////////////
// 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



#include "GTAF.h"
#include "DistfunReg.h" // also includes distdata

namespace mtreeAlgebra {
// en-/disable caching for all node types
const bool nodeCacheEnabled = true;

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

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

Default values for the node config objects (used in the MTreeConfig class):

```
// min. count of pages for leaf / internal nodes
const unsigned minLeafPages   = 1;
const unsigned minIntPages    = 1;

// 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();
```

The following constants should not be changed:

```
using namespace generalTree;
```

```
using gtaf::NodeConfig;
using gtaf::NodeTypeId;

// constants for the node types
const NodeTypeId Leaf = 0;
const NodeTypeId Internal = 1;

// priorities of the node types
const unsigned leafPrio       = 0; // default = 0
const unsigned internalPrio   = 1; // default = 1

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

---

2 Headerfile `MTree.h`

January-March 2008, Mirko Dibbert

2.1 Overview

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

2.2 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
{
```

2.3 Struct **SearchBestPathEntry**:

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

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

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

2.4 Struct **RemainingNodesEntry**:

This struct is used in the `rangeSearch` method of `mtree`.

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

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

2.5 Struct **RemainingNodesEntryNNS**:

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

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

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

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

2.6 Struct **NNEntry**:

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

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

    NNEntry(TupleId _tid, DFUN_RESULT _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;
        }
    }
};
```

2.7 Struct **Header**

```
struct Header : public gtaf::Header
{
    Header() :
        gtaf::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
};
```

2.8 Class **MTree**

```
class MTree : public gtaf::Tree<Header>
```

```
{

public:
```

Default cConstructor, creates a new m-tree.

```
    MTree(bool temporary = false);
```

Constructor, opens an existing tree.

```
    MTree(const SmiFileId fileId);
```

Default copy constructor

```
    MTree(const MTree& mtree);
```

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 `searchRad` to the given `Attribute` object in the result list.

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

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

```
void rangeSearch(DistData* data,
                 const DFUN_RESULT& searchRad,
                 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; }

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 mteeAlgebra
#endif // ifdef __MTREE_H
```

---

3 Headerfile `MTreeBase.h`

January-February 2008, Mirko Dibbert

3.1 Overview

This headerfile implements entries and nodes of the mtree datastructure.

3.2 Includes and defines

```
#ifndef __MTREE_BASE_H
#define __MTREE_BASE_H

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

namespace mtreeAlgebra {
using gtaf::NodePtr;
```

3.3 Deklaration part

3.3.1 Class `LeafEntry`

```
class LeafEntry : public gtaf::LeafEntry
{
friend class InternalEntry;
```

```
public:
```

Default constructor.

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

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

```
inline LeafEntry(TupleId _tid, DistData* _data,
                DFUN_RESULT _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 DFUN_RESULT 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 DFUN_RESULT dist() const
{ return m_dist; }
```

Sets a new distance to parent.

```
inline void setDist(DFUN_RESULT 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 increses offset (defined inline, since this method is called only once from Node::write).

```
inline void write(char* const buffer, int& offset) const
{
    gtaf::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(DFUN_RESULT));
    offset += sizeof(DFUN_RESULT);

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

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

```
inline void read(const char* const buffer, int& offset)
{
    gtaf::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(DFUN_RESULT));
    offset += sizeof(DFUN_RESULT);

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

Returns the size of the entry on disc.

```
inline size_t size()
{
    return gtaf::LeafEntry::size() +
```

```
            sizeof(TupleId) + // m_tid
            sizeof(DFUN_RESULT) +  // 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
    DFUN_RESULT m_dist; // distance to parent node
};
```

3.3.2 Class `InternalEntry`

```
class InternalEntry : public gtaf::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, DFUN_RESULT _rad,
                         SmiRecordId _chield) :
        gtaf::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, DFUN_RESULT _rad,
                         SmiRecordId _chield) :
            gtaf::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 DFUN_RESULT dist() const
    { return m_dist; }
```

Returns the covering radius of the entry.

```
inline DFUN_RESULT 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(DFUN_RESULT dist)
{ m_dist = dist; }
```

Sets a new covering radius.

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

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

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

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

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

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

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

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

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

```
        offset += sizeof(DFUN_RESULT);

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

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

Returns the size of the entry on disc.

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

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

3.3.3 M-Tree basic typedefs

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

typedef SmartPtr<LeafNode> LeafNodePtr;
typedef SmartPtr<InternalNode> InternalNodePtr;

} // namespace mtee_alg
#endif // #ifndef __MTREE_BASE_H
```

4 Headerfile `MTreeConfig.h`

January-February 2008, Mirko Dibbert

4.1 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 by defining the promote and split function, that should be used, as well as the gtaf::NodeConfig objects for internal and leaf nodes, which sets the min/max count entries and pages per node.
All avaliable config objects are defined in the initialize function (file `MTreeConfig.cpp`) and could be set, when using the createmtree2 or createmtree3 operator.

4.2 Includes and defines

```
#ifndef MTREE_CONFIG_H
#define MTREE_CONFIG_H

#include <string>
#include <map>
#include "MTreeSplitpol.h"
#include "MTreeAlgebra.h"

namespace mtreeAlgebra
{

const string CONFIG_DEFAULT("default");
```

4.3 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, 0, 3),
    internalNodeConfig(Internal, 1, 3),
    promoteFun(RANDOM),
    partitionFun(BALANCED)
  {}
```

Constructor (creates objects with the given parameters).

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

4.4 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 the name of the default mtree config.

```
    static inline string defaultName()
    {
        if (!initialized)
            initialize();

        return defaultConfigName;
    }
```

Registeres all `MTreeConfig` objects.

```
    static void initialize();

private:
    static map<string, MTreeConfig> configs;
    static string defaultConfigName;
    static bool initialized;
};

} // namespace mtreeAlgebra

#endif
```

5 Headerfile `MTreeSplitpol.h`

January-February 2008, Mirko Dibbert

5.1 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.

5.2 Includes and defines

```
#ifndef SPLITPOL_H
#define SPLITPOL_H

#include <vector>
#include "MTreeBase.h"

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 should be equal to the parent entry. As second entry, the one with maximum distance to parent 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.

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

```
template<class EntryT>
struct BalancedPromEntry
{
  EntryT* entry;
  DFUN_RESULT distToL, distToR;

  BalancedPromEntry(
      EntryT* entry_, DFUN_RESULT distToL_, DFUN_RESULT distToR_)
  : entry(entry_), distToL(distToL_), distToR(distToR_) {}
};



class Splitpol; // forwar 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 NodeT, class EntryT>
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(SmartPtr<NodeT> _lhs, SmartPtr<NodeT> _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<EntryT*>();
  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<EntryT*>* entries;  // contains the original entry-vector
vector<EntryT*>* entriesL; // new entry vector for left node
vector<EntryT*>* 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

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

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

SmartPtr<NodeT> 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
```

5.3 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
```

5.4 Implementation part for `GenericSplitpol` methods

GenericSplitpol Constructor:

```
  template<class NodeT, class EntryT>
  GenericSplitpol<NodeT, EntryT>::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 NodeT, class EntryT>
void GenericSplitpol<NodeT, EntryT>::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 NodeT, class EntryT>
void GenericSplitpol<NodeT, EntryT>::MRad_Prom()
{
  // precompute distances
  distances = new DFUN_RESULT[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++)
    {
      DFUN_RESULT dist;
      (*metric)((*entries)[i]->data(),
                (*entries)[j]->data(), dist);
      distances[i*entries->size() + j] = dist;
      distances[j*entries->size() + i] = dist;
    }

  bool first = true;
  DFUN_RESULT 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 NodeT, class EntryT>
void GenericSplitpol<NodeT, EntryT>::MMRad_Prom()
{
  // precompute distances
  distances = new DFUN_RESULT[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++)
    {
      DFUN_RESULT dist;
      (*metric)((*entries)[i]->data(),
                (*entries)[j]->data(), dist);
      distances[i*entries->size() + j] = dist;
      distances[j*entries->size() + i] = dist;
    }

  bool first = true;
  DFUN_RESULT 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 NodeT, class EntryT>
void GenericSplitpol<NodeT, EntryT>::MLB_Prom()
{
  #ifdef MTREE_DEBUG
  assert ((*entries)[0]->dist() == 0);
  #endif

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

Method *HyperplanePart* :

```
template<class NodeT, class EntryT>
void GenericSplitpol<NodeT, EntryT>::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
      DFUN_RESULT 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 NodeT, class EntryT>
  void GenericSplitpol<NodeT, EntryT>::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<EntryT> > entriesCpy;
for (size_t i=0; i < entries->size(); i++)
{
  if ((i != promLId) && (i != promRId))
  {
    DFUN_RESULT 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<EntryT> (((*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<EntryT> >::iterator
        nearestPos = entriesCpy.begin();

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

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

```
      DFUN_RESULT 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<EntryT> >::iterator
          nearestPos = entriesCpy.begin();

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

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

6 Implementation file `MTreeAlgebra.cpp`

January-March 2008, Mirko Dibbert

6.1 Overview

This file contains the implementation of the mtree algebra.

6.2 Includes and defines

```
#include "Algebra.h"
#include "NestedList.h"
#include "QueryProcessor.h"
#include "RelationAlgebra.h"
#include "TupleIdentifier.h"
#include "MTreeAlgebra.h"
#include "MTree.h"

extern NestedList* nl;
extern QueryProcessor* qp;
extern AlgebraManager* am;

namespace mtreeAlgebra {
```

6.3 Type constructor `MTREE`

```
static ListExpr
```

```
MTreeProp()
{
    ListExpr examplelist = nl->TextAtom();
    nl->AppendText(examplelist, "<relation> createmtree [<attrname>]"
                                     " where <attrname> is the key");

    return (nl->TwoElemList(
            nl->TwoElemList(nl->StringAtom("Creation"),
                    nl->StringAtom("Example Creation")),
            nl->TwoElemList(examplelist,
                    nl->StringAtom("(let mymtree = images "
                                    "createmtree[pic] "))));
}

ListExpr
OutMTree(ListExpr type_Info, Word w)
{
    MTree* mtree = static_cast<MTree*>(w.addr);
    if (mtree->isInitialized())
    {
        NList assignments(
            NList("assignments:"),
            NList(
                NList(NList("type constructor:"),
                    NList(mtree->typeName(), true)),
                NList(NList("metric:"),
                    NList(mtree->distfunName(), true)),
                NList(NList("distdata type:"),
                    NList(mtree->dataName(), true)),
                NList(NList("mtree-config:"),
                    NList(mtree->configName(), true))));

        NList statistics(
            NList("statistics:"),
            NList(
                NList(NList("height:"),
                    NList((int)mtree->height())),
                NList(NList("# of internal nodes:"),
                    NList((int)mtree->internalCount())),
                NList(NList("# of leaf nodes:"),
                    NList((int)mtree->leafCount())),
                NList(NList("# of leaf entries:"),
                    NList((int)mtree->entryCount()))));

        NList result(assignments, statistics);
        return result.listExpr();
    }
    else
        return nl->SymbolAtom( "undef" );
}

Word
InMTree(ListExpr type_Info, ListExpr value,
```

```
        int errorPos, ListExpr &error_Info, bool &correct)
{
    correct = false;
    return SetWord(0);
}

Word
Createmtree(const ListExpr type_Info)
{ return SetWord(new MTree()); }

void
DeleteMTree(const ListExpr type_Info, Word& w)
{
    static_cast<MTree*>(w.addr)->deleteFile();
    delete static_cast<MTree*>(w.addr);
    w.addr = 0;
}

bool
OpenMTree(SmiRecord &valueRecord, size_t &offset,
          const ListExpr type_Info, Word& w)
{
    SmiFileId fileid;
    valueRecord.Read(&fileid, sizeof(SmiFileId), offset);
    offset += sizeof(SmiFileId);

    MTree* mtree = new MTree(fileid);
    w = SetWord(mtree);
    return true;
}

bool
SaveMTree(SmiRecord &valueRecord, size_t &offset,
          const ListExpr type_Info, Word& w)
{
    SmiFileId fileId;
    MTree *mtree = static_cast<MTree*>(w.addr);
    fileId = mtree->fileId();
    if (fileId)
    {
        valueRecord.Write(&fileId, sizeof(SmiFileId), offset);
        offset += sizeof(SmiFileId);
        return true;
    }
    else
    {
        return false;
    }
}

void
CloseMTree(const ListExpr type_Info, Word& w)
{
```

```
    MTree *mtree = (MTree*)w.addr;
    delete mtree;
}

Word
CloneMTree(const ListExpr type_Info, const Word& w)
{
    MTree* src = static_cast<MTree*>(w.addr);
    MTree* cpy = new MTree(*src);
    return SetWord(cpy);
}

int
SizeOfMTree()
{ return sizeof(MTree); }

bool
CheckMTree(ListExpr typeName, ListExpr &error_Info)
{ return nl->IsEqual(typeName, MTREE); }

TypeConstructor
mtreeTC(MTREE,       MTreeProp,
        OutMTree,    InMTree,
        0, 0,
        Createmtree, DeleteMTree,
        OpenMTree,   SaveMTree,
        CloseMTree,  CloneMTree,
        0,
        SizeOfMTree,
        CheckMTree);
```

6.4 Operators

6.4.1 Value mappings

6.4.1.1 createmtreeRel_VM This value mapping function is used for all `createmtree` operators, which expect a relation and non-distdata attributes.

It is designed as template function, which expects the count of arguments as template paremeter.

```
template<unsigned paramCnt> int
        createmtreeRel_VM(Word* args, Word& result, int message,
                          Word& local, Supplier s)
{
    result = qp->ResultStorage(s);

    MTree* mtree =
        static_cast<MTree*>(result.addr);

    Relation* relation =
        static_cast<Relation*>(args[0].addr);
```

```
    int attrIndex =
        static_cast<CcInt*>(args[paramCnt].addr)->GetIntval();

    string typeName =
        static_cast<CcString*>(args[paramCnt+1].addr)->GetValue();

    string distfunName =
        static_cast<CcString*>(args[paramCnt+2].addr)->GetValue();

    string dataName =
        static_cast<CcString*>(args[paramCnt+3].addr)->GetValue();

    string configName =
        static_cast<CcString*>(args[paramCnt+4].addr)->GetValue();

    DistDataId id = DistDataReg::getDataId(typeName, dataName);
    mtree->initialize(id, distfunName, configName);

    Tuple* tuple;
    GenericRelationIterator* iter = relation->MakeScan();
    while ((tuple = iter->GetNextTuple()) != 0)
    {
        Attribute* attr = tuple->GetAttribute(attrIndex);
        if(attr->IsDefined())
        {
            mtree->insert(attr, tuple->GetTupleId());
        }
        tuple->DeleteIfAllowed();
    }
    delete iter;

    #ifdef MTREE_PRINT_INSERT_INFO
    cout << endl;
    #endif

    return 0;
}
```

6.4.1.2 createmtreeStream_VM This value mapping function is used for all `createmtree` operators, which expect a tupel stream and non-distdata attributes.

It is designed as template function, which expects the count of arguments as template paremeter.

```
template<unsigned paramCnt> int
createmtreeStream_VM(Word* args, Word& result, int message,
                     Word& local, Supplier s)
{
    result = qp->ResultStorage(s);
    MTree* mtree = static_cast<MTree*>(result.addr);

    void* stream =
        static_cast<Relation*>(args[0].addr);
```

```
    int attrIndex =
        static_cast<CcInt*>(args[paramCnt].addr)->GetIntval();

    string typeName =
        static_cast<CcString*>(args[paramCnt+1].addr)->GetValue();

    string distfunName =
        static_cast<CcString*>(args[paramCnt+2].addr)->GetValue();

    string dataName =
        static_cast<CcString*>(args[paramCnt+3].addr)->GetValue();

    string configName =
        static_cast<CcString*>(args[paramCnt+4].addr)->GetValue();

    DistDataId id = DistDataReg::getDataId(typeName, dataName);
    mtree->initialize(id, distfunName, configName);

    Word wTuple;
    qp->Open(stream);
    qp->Request(stream, wTuple);
    while (qp->Received(stream))
    {
        Tuple* tuple = static_cast<Tuple*>(wTuple.addr);
        Attribute* attr = tuple->GetAttribute(attrIndex);
        if(attr->IsDefined())
        {
            mtree->insert(attr, tuple->GetTupleId());
        }
        tuple->DeleteIfAllowed();
        qp->Request(stream, wTuple);
    }
    qp->Close(stream);

    #ifdef MTREE_PRINT_INSERT_INFO
    cout << endl;
    #endif

    return 0;
}
```

6.4.1.3 createmtreeDDRel_VM

This value mapping function is used for all `createmtree` operators, which expect a relation and distdata attributes.

It is designed as template function, which expects the count of arguments as template paremeter.

```
template<unsigned paramCnt> int
createmtreeDDRel_VM(Word* args, Word& result, int message,
                    Word& local, Supplier s)
{
    result = qp->ResultStorage(s);
    MTree* mtree = static_cast<MTree*>(result.addr);
```

```
Relation* relation =
    static_cast<Relation*>(args[0].addr);

int attrIndex =
    static_cast<CcInt*>(args[paramCnt].addr)->GetIntval();

string distfunName =
    static_cast<CcString*>(args[paramCnt+1].addr)->GetValue();

string configName =
    static_cast<CcString*>(args[paramCnt+2].addr)->GetValue();

Tuple* tuple;
GenericRelationIterator* iter = relation->MakeScan();

DistDataId id;
while ((tuple = iter->GetNextTuple()))
{
    DistDataAttribute* attr = static_cast<DistDataAttribute*>(
            tuple->GetAttribute(attrIndex));

    if(attr->IsDefined())
    {
        if (mtree->isInitialized())
        {
            if(attr->distdataId() != id)
            {
                const string seperator =
                        "\n" + string(70, '-') + "\n";
                cmsg.error()
                    << seperator
                    << "Operator createmtree: " << endl
                    << "Got distdata attributes of different "
                    << "types!" << endl << "(type constructor "
                    << "or distdata type are not equal)"
                    << seperator << endl;
                cmsg.send();
                tuple->DeleteIfAllowed();
                delete iter;
                return CANCEL;
            }
        }
        else
        {  // initialize mtree
            id = attr->distdataId();
            DistDataInfo info = DistDataReg::getInfo(id);
            string dataName = info.name();
            string typeName = info.typeName();

            if (distfunName == DFUN_DEFAULT)
            {
                distfunName = DistfunReg::defaultName(typeName);
            }
```

```
            // check if distance function is defined
            if (!DistfunReg::isDefined(distfunName, id))
            {
                const string seperator =
                        "\n" + string(70, '-') + "\n";
                cmsg.error()
                    << seperator
                    << "Operator createmtree: " << endl
                    << "Distance function \"" << distfunName
                    << "\" for type \"" << typeName
                    << "\" is not defined!" << endl
                    << "Defined distance functions: " << endl
                    << endl
                    << DistfunReg::definedNames(typeName)
                    << seperator << endl;
                cmsg.send();
                tuple->DeleteIfAllowed();
                delete iter;
                return CANCEL;
            }

            if (!DistfunReg::getInfo(
                    distfunName, typeName, dataName).isMetric())
            {
                const string seperator =
                        "\n" + string(70, '-') + "\n";
                cmsg.error()
                    << seperator
                    << "Operator createmtree: " << endl
                    << "Distance function \"" << distfunName
                    << "\" with \"" << dataName
                    << "\" data for type" << endl
                    << "\"" << typeName << "\" is no metric!"
                    << seperator << endl;
                cmsg.send();
                tuple->DeleteIfAllowed();
                delete iter;
                return CANCEL;
            }

            mtree->initialize(
                    attr->distdataId(), distfunName, configName);
        }

        // insert attribute into mtree
        DistData* data =
                new DistData(attr->size(), attr->value());
        mtree->insert(data, tuple->GetTupleId());
    }
    tuple->DeleteIfAllowed();
}
delete iter;
```

```
    #ifdef MTREE_PRINT_INSERT_INFO
    cout << endl;
    #endif

    return 0;
}
```

6.4.1.4 createmtreeDDStream_VM This value mapping function is used for all `createmtree` operators, which expect a tupel stream and distdata attributes.

It is designed as template function, which expects the count of arguments as template paremeter.

```
template<unsigned paramCnt> int
createmtreeDDStream_VM(Word* args, Word& result, int message,
                       Word& local, Supplier s)
{
    result = qp->ResultStorage(s);
    MTree* mtree = static_cast<MTree*>(result.addr);

    void* stream =
        static_cast<Relation*>(args[0].addr);

    int attrIndex =
        static_cast<CcInt*>(args[paramCnt].addr)->GetIntval();

    string distfunName =
        static_cast<CcString*>(args[paramCnt+1].addr)->GetValue();

    string configName =
        static_cast<CcString*>(args[paramCnt+2].addr)->GetValue();

    DistDataId id;
    Word wTuple;
    qp->Open(stream);
    qp->Request(stream, wTuple);
    while (qp->Received(stream))
    {
        Tuple* tuple = static_cast<Tuple*>(wTuple.addr);
        DistDataAttribute* attr = static_cast<DistDataAttribute*>(
                                     tuple->GetAttribute(attrIndex));
        if(attr->IsDefined())
        {
            if (mtree->isInitialized())
            {
                if(attr->distdataId() != id)
                {
                    const string seperator =
                            "\n" + string(70, '-') + "\n";
                    cmsg.error()
                        << seperator
                        << "Operator createmtree: " << endl
                        << "Got distdata attributes of different "
```

```
            << "types!" << endl << "(type constructor "
            << "or distdata type are not equal)"
            << seperator << endl;
        cmsg.send();
        tuple->DeleteIfAllowed();
        return CANCEL;
    }
}
else
{  // initialize mtree
    id = attr->distdataId();
    DistDataInfo info = DistDataReg::getInfo(id);
    string dataName = info.name();
    string typeName = info.typeName();

    if (distfunName == DFUN_DEFAULT)
    {
        distfunName = DistfunReg::defaultName(typeName);
    }

    // check if distance function is defined
    if (!DistfunReg::isDefined(distfunName, id))
    {
        const string seperator =
                "\n" + string(70, '-') + "\n";
        cmsg.error()
            << seperator
            << "Operator createmtree: " << endl
            << "Distance function \"" << distfunName
            << "\" for type \"" << typeName
            << "\" is not defined!" << endl
            << "Defined distance functions: " << endl
            << endl
            << DistfunReg::definedNames(typeName)
            << seperator << endl;
        cmsg.send();
        tuple->DeleteIfAllowed();
        return CANCEL;
    }

    if (!DistfunReg::getInfo(
            distfunName, typeName, dataName).isMetric())
    {
        const string seperator =
                "\n" + string(70, '-') + "\n";
        cmsg.error()
            << seperator
            << "Operator createmtree: " << endl
            << "Distance function \"" << distfunName
            << "\" with \"" << dataName
            << "\" data for type" << endl
            << "\"" << typeName << "\" is no metric!"
            << seperator << endl;
```

```
                    cmsg.send();
                    tuple->DeleteIfAllowed();
                    return CANCEL;
                }

                mtree->initialize(
                        attr->distdataId(), distfunName, configName);
            }

            // insert attribute into mtree
            DistData* data =
                    new DistData(attr->size(), attr->value());
            mtree->insert(data, tuple->GetTupleId());
        }
        tuple->DeleteIfAllowed();
        qp->Request(stream, wTuple);
    }

    qp->Close(stream);

    #ifdef MTREE_PRINT_INSERT_INFO
    cout << endl;
    #endif

    return 0;
}
```

6.4.1.5 rangesearchLocalInfo

```
struct rangesearchLocalInfo
{
  Relation* relation;
  list<TupleId>* results;
  list<TupleId>::iterator iter;
  bool defined;

  rangesearchLocalInfo(Relation* rel) :
    relation(rel),
    results(new list<TupleId>),
    defined(false)
    {}

  void initResultIterator()
  {
    iter = results->begin();
    defined = true;
  }

  ~rangesearchLocalInfo()
  {
    delete results;
  }
```

```
  TupleId next()
  {
    if (iter != results->end())
    {
      TupleId tid = *iter;
      *iter++;
      return tid;
    }
    else
    {
      return 0;
    }
  }
};
```

6.4.1.6 rangesearch_VM

```
int
rangesearch_VM(Word* args, Word& result, int message,
               Word& local, Supplier s)
{
  rangesearchLocalInfo* info;

  switch (message)
  {
    case OPEN :
    {
      MTree* mtree =
          static_cast<MTree*>(args[0].addr);

      info = new rangesearchLocalInfo(
          static_cast<Relation*>(args[1].addr));
      local = SetWord(info);

      Attribute* attr =
          static_cast<Attribute*>(args[2].addr);

      double searchRad =
            static_cast<CcReal*>(args[3].addr)->GetValue();

      string typeName =
          static_cast<CcString*>(args[4].addr)->GetValue();

      if (mtree->typeName() != typeName)
      {
        const string seperator = "\n" + string(70, '-') + "\n";
        cmsg.error() << seperator
            << "Operator rangesearch:" << endl
            << "Got an \"" << typeName << "\" attribute, but the "
            << "mtree contains \"" << mtree->typeName()
            << "\" attriubtes!" << seperator << endl;
        cmsg.send();
        return CANCEL;
```

```
      }

      mtree->rangeSearch(attr, searchRad, info->results);
      info->initResultIterator();

      assert(info->relation != 0);
      return 0;
    }

    case REQUEST :
    {
      info = (rangesearchLocalInfo*)local.addr;
      if(!info->defined)
        return CANCEL;

      TupleId tid = info->next();
      if(tid)
      {
        Tuple *tuple = info->relation->GetTuple(tid);
        result = SetWord(tuple);
        return YIELD;
      }
      else
      {
        return CANCEL;
      }
    }

    case CLOSE :
    {
      info = (rangesearchLocalInfo*)local.addr;
      delete info;
      return 0;
    }
  }

  return 0;
}
```

6.4.1.7 rangesearchDD_VM

```
int
rangesearchDD_VM(Word* args, Word& result, int message,
                 Word& local, Supplier s)
{
  rangesearchLocalInfo* info;

  switch (message)
  {
    case OPEN :
    {
      MTree* mtree = static_cast<MTree*>(args[0].addr);
```

```
  info = new rangesearchLocalInfo(
      static_cast<Relation*>(args[1].addr));
  local = SetWord(info);

  DistDataAttribute* attr =
      static_cast<DistDataAttribute*>(args[2].addr);

  double searchRad =
        static_cast<CcReal*>(args[3].addr)->GetValue();

  string typeName =
      static_cast<CcString*>(args[4].addr)->GetValue();

  if (attr->distdataId() != mtree->dataId())
  {
    const string seperator = "\n" + string(70, '-') + "\n";
    cmsg.error() << seperator
      << "Operator rangesearch:" << endl
      << "Distdata attribute type does not match the type of "
      << "the mtree!" << seperator << endl;
    cmsg.send();
    return CANCEL;
  }

  DistData* data = new DistData(attr->size(), attr->value());
  mtree->rangeSearch(data, searchRad, info->results);
  info->initResultIterator();

  assert(info->relation != 0);
  return 0;
}

case REQUEST :
{
  info = (rangesearchLocalInfo*)local.addr;
  if(!info->defined)
    return CANCEL;

  TupleId tid = info->next();
  if(tid)
  {
    Tuple *tuple = info->relation->GetTuple(tid);
    result = SetWord(tuple);
    return YIELD;
  }
  else
  {
    return CANCEL;
  }
}

case CLOSE :
{
```

```
      info = (rangesearchLocalInfo*)local.addr;
      delete info;
      return 0;
    }
  }
  return 0;
}
```

6.4.1.8 nnsearchLocalInfo

```
struct nnsearchLocalInfo
{
  Relation* relation;
  list<TupleId>* results;
  list<TupleId>::iterator iter;
  bool defined;

  nnsearchLocalInfo(Relation* rel)
  : relation(rel),
    results(new list<TupleId>),
    defined(false)
    {}

  void initResultIterator()
  {
    iter = results->begin();
    defined = true;
  }

  ~nnsearchLocalInfo()
  {
    delete results;
  }

  TupleId next()
  {
    if (iter != results->end())
    {
      TupleId tid = *iter;
      *iter++;
      return tid;
    }
    else
    {
      return 0;
    }
  }
};
```

6.4.1.9 nnsearch_VM

```
int
```

```
nnsearch_VM(Word* args, Word& result, int message,
            Word& local, Supplier s)
{
  nnsearchLocalInfo* info;

  switch (message)
  {
    case OPEN :
    {
      MTree* mtree =
          static_cast<MTree*>(args[0].addr);

      info = new nnsearchLocalInfo(
          static_cast<Relation*>(args[1].addr));
      local = SetWord(info);

      Attribute* attr =
          static_cast<Attribute*>(args[2].addr);

      int nncount= ((CcInt*)args[3].addr)->GetValue();

      string typeName =
          static_cast<CcString*>(args[4].addr)->GetValue();

      if (mtree->typeName() != typeName)
      {
        const string seperator = "\n" + string(70, '-') + "\n";
        cmsg.error() << seperator
            << "Operator nnsearch:" << endl
            << "Got an \"" << typeName << "\" attribute, but the "
            << "mtree contains \"" << mtree->typeName()
            << "\" attriubtes!" << seperator << endl;
        cmsg.send();
        return CANCEL;
      }

      mtree->nnSearch(attr, nncount, info->results);
      info->initResultIterator();

      assert(info->relation != 0);
      return 0;
    }

    case REQUEST :
    {
      info = (nnsearchLocalInfo*)local.addr;
      if(!info->defined)
        return CANCEL;

      TupleId tid = info->next();
      if(tid)
      {
        Tuple *tuple = info->relation->GetTuple(tid);
```

```
        result = SetWord(tuple);
        return YIELD;
      }
      else
      {
        return CANCEL;
      }
    }

    case CLOSE :
    {
      info = (nnsearchLocalInfo*)local.addr;
      delete info;
      return 0;
    }
  }
  return 0;
}
```

6.4.1.10 nnsearchDD_VM

```
int
nnsearchDD_VM(Word* args, Word& result, int message,
              Word& local, Supplier s)
{
  nnsearchLocalInfo* info;
  switch (message)
  {
    case OPEN :
    {
      MTree* mtree = static_cast<MTree*>(args[0].addr);

      info = new nnsearchLocalInfo(
          static_cast<Relation*>(args[1].addr));
      local = SetWord(info);

      DistDataAttribute* attr =
          static_cast<DistDataAttribute*>(args[2].addr);

      int nncount = ((CcInt*)args[3].addr)->GetValue();


      if (attr->distdataId() != mtree->dataId())
      {
        const string seperator = "\n" + string(70, '-') + "\n";
        cmsg.error() << seperator
          << "Operator nnsearch:" << endl
          << "Distdata attribute type does not match the type of "
          << "the mtree!" << seperator << endl;
        cmsg.send();
        return CANCEL;
      }
```

```
      DistData* data = new DistData(attr->size(), attr->value());
      mtree->nnSearch(data, nncount, info->results);
      info->initResultIterator();

      assert(info->relation != 0);
      return 0;
    }

    case REQUEST :
    {
      info = (nnsearchLocalInfo*)local.addr;
      if(!info->defined)
         return CANCEL;

      TupleId tid = info->next();
      if(tid)
      {
        Tuple *tuple = info->relation->GetTuple(tid);
        result = SetWord(tuple);
        return YIELD;
      }
      else
      {
        return CANCEL;
      }
    }

    case CLOSE :
    {
      info = (nnsearchLocalInfo*)local.addr;
      delete info;
      return 0;
    }
  }
  return 0;
}
```

6.4.2 Type mappings

6.4.2.1 createmtree_TM relation/tuple stream x attribute name -¿ mtree (op. createmtree)
relation/tuple stream x attribute name x config name x distfun name -¿ mtree (op. createmtree2)
relation/tuple stream x attribute name x config name x distfun name x distdata type -¿ mtree (op. createmtree3)

```
template<unsigned paramCnt>
ListExpr createmtree_TM(ListExpr args)
{
    // initialize distance functions and distdata types
    if (!DistfunReg::isInitialized())
        DistfunReg::initialize();
```

```
stringstream paramCntErr;
string errmsg;
bool cond;
NList nl_args(args);

paramCntErr << "Expecting " << paramCnt << " arguments.";
cond = nl_args.length() == paramCnt;
CHECK_COND(cond, paramCntErr.str());

NList arg1 = nl_args.first();
NList arg2 = nl_args.second();

// check first argument (should be relation or tuple stream)
NList attrs;
cond = (arg1.checkRel(attrs) || arg1.checkStreamTuple(attrs));
errmsg = "Expecting a relation or tuple stream as first "
         "argument, but got a list with structure '" +
          arg1.convertToString() + "'.";
CHECK_COND(cond, errmsg);

// check, if second argument is the name of an existing attribute
errmsg = "Expecting the name of an existing attribute as second "
         "argument, but got '" + arg2.convertToString() + "'.";
CHECK_COND(arg2.isSymbol(), errmsg);

// check, if attribute can be found in attribute list
string attrName = arg2.str();
errmsg = "Attribute name '" + attrName + "' is not known.\n"
         "Known Attribute(s):\n" + attrs.convertToString();
ListExpr attrTypeLE;
int attrIndex = FindAttribute(
        attrs.listExpr(), attrName, attrTypeLE);
CHECK_COND(attrIndex > 0, errmsg);
NList attrType (attrTypeLE);
string typeName = attrType.str();

// select config name
string configName;
if (paramCnt >= 3)
{  // type mapping for createmtree2 and createmtree3
    NList arg3 = nl_args.third();
    errmsg = "Expecting the name of an existing mtree config "
             "or '" + CONFIG_DEFAULT + "\" as third argument.";
    CHECK_COND(arg3.isSymbol(), errmsg);
    configName = arg3.str();
}
else
{  // type mapping for createmtree
    configName = CONFIG_DEFAULT;
}

// check, if selected config name is defined
if (configName == CONFIG_DEFAULT)
```

```
{
    configName = MTreeConfigReg::defaultName();
    errmsg = "Default config (\"" + configName +
             "\") not defined!";
}
else
{
    errmsg = "Config \"" + configName + "\" not defined!";
}
CHECK_COND(MTreeConfigReg::isDefined(configName), errmsg);

// select distfun name
string distfunName;
if (paramCnt >= 4)
{  // type mapping for createmtree2 and createmtree3
    NList arg4 = nl_args.fourth();
    errmsg = "Expecting the name of an existing distance function "
             "or '" + DFUN_DEFAULT + "\" as fourth argument.";
    CHECK_COND(arg4.isSymbol(), errmsg);
    distfunName = arg4.str();
}
else
{  // type mapping for createmtree
    distfunName = DFUN_DEFAULT;
}

if (typeName == DISTDATA)
{
    NList res1(APPEND);
    NList res2;
    res2.append(NList(attrIndex - 1));
    res2.append(NList(distfunName, true));
    res2.append(NList(configName, true));
    NList res3(MTREE);
    NList result(res1, res2, res3);
    return result.listExpr();
}

// *** typeName != DISTDATA ***

// select distdata type
string dataName;
if (paramCnt == 5)
{ // type mapping for createmtree3
    NList arg5 = nl_args.fifth();
    errmsg = "Expecting the name of an existing distdata type "
             "or '" + DDATA_DEFAULT + "\" as fifth argument.";
    CHECK_COND(arg5.isSymbol(), errmsg);
    dataName = arg5.str();
}
else
{ // type mapping for createmtree1 and createmtree2
    dataName = DistDataReg::defaultName(typeName);
```

```
}

// check, if selected distance function with selected distdata
// type is defined
if (dataName == DDATA_DEFAULT)
{
    errmsg = "No default distdata type defined for type \"" +
             typeName + "\"!";
    dataName = DistDataReg::defaultName(typeName);
    CHECK_COND(dataName != DDATA_UNDEFINED, errmsg);
}
else if(!DistDataReg::isDefined(typeName, dataName))
{
    errmsg = "Distdata type \"" + dataName + "\" for type \"" +
             typeName + "\" is not defined! Defined names: \n\n" +
             DistDataReg::definedNames(typeName);
    CHECK_COND(false, errmsg);
}

if (distfunName == DFUN_DEFAULT)
{
    distfunName = DistfunReg::defaultName(typeName);
    errmsg = "No default distance function defined for type \""
        + typeName + "\"!";
    CHECK_COND(distfunName != DFUN_UNDEFINED, errmsg);
}
else
{ // search distfun
    if (!DistfunReg::isDefined(
            distfunName, typeName, dataName))
    {
        errmsg = "Distance function \"" + distfunName +
                 "\" not defined for type \"" +
                 typeName + "\" and data type \"" +
                 dataName + "\"! Defined names: \n\n" +
                 DistfunReg::definedNames(typeName);
        CHECK_COND(false, errmsg);
    }
}

// check if selected distance function is a metric
errmsg = "Distance function \"" + distfunName +
         "\" with \"" + dataName + "\" data for type \"" +
         typeName + "\" is no metric!";
cond = DistfunReg::getInfo(
        distfunName, typeName, dataName).isMetric();
CHECK_COND(cond, errmsg);

// generate result list
NList res1(APPEND);
NList res2;
res2.append(NList(attrIndex - 1));
res2.append(NList(typeName, true));
```

```
    res2.append(NList(distfunName, true));
    res2.append(NList(dataName, true));
    res2.append(NList(configName, true));
    NList res3(MTREE);
    NList result(res1, res2, res3);

    return result.listExpr();
}
```

6.4.2.2 rangesearch_TM

```
ListExpr
rangesearch_TM(ListExpr args)
{
    // initialize distance functions and distdata types
    if (!DistfunReg::isInitialized())
        DistfunReg::initialize();

  string errmsg;
  NList nl_args(args);

  errmsg = "Operator rangesearch expects four arguments(mtree x "
           "relation x search_attribute x search_range)";
  CHECK_COND(nl_args.length() == 4, errmsg);

  NList arg1 = nl_args.first();
  NList arg2 = nl_args.second();
  NList arg3 = nl_args.third();
  NList arg4 = nl_args.fourth();

  // check first argument (should be a mtree)
  errmsg = "Expecting a mtree as first argument!";
  CHECK_COND(arg1.isEqual(MTREE), errmsg);

  // check second argument (should be relation)
  NList attrs;
  errmsg = "Expecting a relation as second argument, but got a "
           "list with structure '" + arg2.convertToString() + "'.";
  CHECK_COND(arg2.checkRel(attrs), errmsg);

  // check fourth argument
  errmsg = "Expecting an int value as fourth argument, but got '" +
           arg4.convertToString() + "'.";
  CHECK_COND(arg4.isEqual(REAL), errmsg);

  NList append(APPEND);
  NList result (
      append,
      NList(arg3.convertToString(), true).enclose(),
      NList(NList(STREAM), arg2.second()));
  return result.listExpr();
}
```

6.4.2.3 nnsearch_TM

```
ListExpr
nnsearch_TM(ListExpr args)
{
    // initialize distance functions and distdata types
    if (!DistfunReg::isInitialized())
        DistfunReg::initialize();

  string errmsg;
  NList nl_args(args);

  errmsg = "Operator nnsearch expects four arguments(mtree x "
           "relation x search_attribute x nncount)";
  CHECK_COND(nl_args.length() == 4, errmsg);

  NList arg1 = nl_args.first();
  NList arg2 = nl_args.second();
  NList arg3 = nl_args.third();
  NList arg4 = nl_args.fourth();

  // check first argument (should be a mtree)
  errmsg = "Expecting a mtree as first argument!";
  CHECK_COND(arg1.isEqual(MTREE), errmsg);

  // check second argument (should be relation)
  NList attrs;
  errmsg = "Expecting a list with structure\n"
           "   rel (tuple ((a1 t1)...(an tn)))\n"
           "as second argument, but got a list with structure '" +
       arg2.convertToString() + "'.";
  CHECK_COND(arg2.checkRel(attrs), errmsg);

  // check fourth argument
  errmsg = "Expecting an int value as fourth argument, but got '" +
           arg4.convertToString() + "'.";
  CHECK_COND(arg4.isEqual(INT), errmsg);

  NList append(APPEND);
  NList result (
      append,
      NList(arg3.convertToString(), true).enclose(),
      NList(NList(STREAM), arg2.second()));
  return result.listExpr();
}
```

6.4.3 Selection functions

```
int
createmtree_Select(ListExpr args)
{
    NList argsNL(args);
    NList arg1 = argsNL.first();
```

```
    NList attrs = arg1.second().second();
    NList arg2 = argsNL.second();

    // get type of selected attribute
    string attrName = arg2.str();
    ListExpr attrTypeLE;
    FindAttribute(attrs.listExpr(), attrName, attrTypeLE);
    NList attrType(attrTypeLE);

    if (arg1.first().isEqual(REL))
    {
        if(attrType.isEqual(DISTDATA))
            return 2;
        else
            return 0;
    }
    else if (arg1.first().isEqual(STREAM))
    {
        if(attrType.isEqual(DISTDATA))
            return 3;
        else
            return 1;
    }
    else
        return -1;
}

int
createmtree3_Select(ListExpr args)
{
    NList argsNL(args);
    NList arg1 = argsNL.first();

    if (arg1.first().isEqual(REL))
        return 0;
    else if (arg1.first().isEqual(STREAM))
        return 1;
    else
        return -1;
}

int
search_Select(ListExpr args)
{
    NList argsNL(args);
    NList arg3 = argsNL.third();
    if (arg3.isEqual(DISTDATA))
        return 1;
    else
        return 0;
}
```

6.4.4 Value mapping arrays

```
ValueMapping createmtree_Map[] = {
    createmtreeRel_VM<2>,
    createmtreeStream_VM<2>,
    createmtreeDDRel_VM<2>,
    createmtreeDDStream_VM<2>
};

ValueMapping createmtree2_Map[] = {
    createmtreeRel_VM<4>,
    createmtreeStream_VM<4>,
    createmtreeDDRel_VM<4>,
    createmtreeDDStream_VM<4>
};

ValueMapping createmtree3_Map[] = {
    createmtreeRel_VM<5>,
    createmtreeStream_VM<5>,
    createmtreeDDRel_VM<5>,
    createmtreeDDStream_VM<5>
};

ValueMapping rangesearch_Map[] = {
    rangesearch_VM,
    rangesearchDD_VM
};

ValueMapping nnsearch_Map[] = {
    nnsearch_VM,
    nnsearchDD_VM
};
```

6.4.5 Operator infos

```
struct createmtree_Info : OperatorInfo
{
    createmtree_Info()
    {
        name = "createmtree";
        signature =
        "(<text>(rel (tuple ((id tid) (x1 t1)...(xn tn)))"
        " metricName, xi) -> mtree";
        syntax = "_ createmtree [_]";
        meaning =
            "creates a new mtree from relation or tuple stream in arg1\n"
            "arg2 must be the name of the attribute in arg1, "
            "which should be indexed by the mtree";
        example = "pictures createmtree [Pic]";
    }
};
```

```
struct createmtree2_Info : OperatorInfo
{
    createmtree2_Info()
    {
        name = "createmtree2";
        signature =
        "(<text>(rel (tuple ((id tid) (x1 t1)...(xn tn)))"
        " metricName, xi) -> mtree";
        syntax = "_ createmtree2 [_, _, _]";
        meaning =
            "creates a new mtree from relation or tuple stream in arg1\n"
            "arg2 must be the name of the attribute in arg1, "
            "which should be indexed by the mtree\n"
            "arg3 must be the name of a registered mtree-config\n"
            "arg4 must be the name of a registered metric";
        example = "pictures createmtree2 [Pic, mlbdistHP, quadr]";
    }
};

struct createmtree3_Info : OperatorInfo
{
    createmtree3_Info()
    {
        name = "createmtree3";
        signature =
        "(<text>(rel (tuple ((id tid) (x1 t1)...(xn tn)))"
        "metric, mtreeconfig, xi) -> mtree";
        syntax = "_ createmtree3 [_, _, _, _]";
        meaning =
            "creates a new mtree from relation or tuple stream in arg1\n"
            "arg2 must be the name of the attribute in arg1, "
            "which should be indexed by the mtree\n"
            "arg3 must be the name of a registered mtree-config\n"
            "arg4 must be the name of a registered metric\n"
            "arg5 must be the name of a registered distdata type";
        example = "pictures createmtree [lab, mlbdistHP, quadr, lab256]";
    }
};

struct rangesearch_Info : OperatorInfo
{
    rangesearch_Info()
    {
        name = "rangesearch";
        signature =
        "(<text>mtree x (rel (tuple ((id tid) (x1 t1)...(xn tn)))) "
        "x attribute x real -> "
        "(stream (tuple ((x1 t1)...(xn tn))))";
        syntax = "_ rangesearch [_, _, _]";
        meaning = "arg1: mtree\n"
                "arg2: relation, that must contain at "
                "least all tuple id’s that are indized in the mtree\n"
                "arg3: reference attribute\n"
```

```
                "arg4: maximum distance to arg3";
        example = "pictree rangesearch [pictures, pic1, 0.2]";
    }
};

struct nnsearch_Info : OperatorInfo
{
    nnsearch_Info()
    {
        name = "nnsearch";
        signature =
        "(<text>mtree x (rel (tuple ((id tid) (x1 t1)...(xn tn)))) "
        "x attribute x int -> "
        "(stream (tuple ((x1 t1)...(xn tn))))";
        syntax = "_ nnsearch [_, _, _]";
        meaning = "arg1: mtree\n"
                "arg2: relation, that must contain at "
                "least all tuple id's that are indized in the mtree\n"
                "arg3: reference attribute\n"
                "arg4: the count of nearest neighbours of arg3 "
                "which should be returned";

        example = "pictree nnsearch [pictures, pic1, 5]";
    }
};
```

6.5 Create and initialize the Algebra

```
class MTreeAlgebra : public Algebra
{

public:
    MTreeAlgebra() : Algebra()
    {
        AddTypeConstructor(&mtreeTC);

        AddOperator(createmtree_Info(),
                    createmtree_Map,
                    createmtree_Select,
                    createmtree_TM<2>);

        AddOperator(createmtree2_Info(),
                    createmtree2_Map,
                    createmtree_Select,
                    createmtree_TM<4>);

        AddOperator(createmtree3_Info(),
                    createmtree3_Map,
                    createmtree_Select,
                    createmtree_TM<5>);

        AddOperator(rangesearch_Info(),
```

```
                    rangesearch_Map,
                    search_Select,
                    rangesearch_TM);

        AddOperator(nnsearch_Info(),
                    nnsearch_Map,
                    search_Select,
                    nnsearch_TM);
    }

    ~MTreeAlgebra() {};
};

} // namespace mtreeAlgebra

mtreeAlgebra::MTreeAlgebra mtreeAlg;

extern "C"
Algebra* InitializeMTreeAlgebra(
    NestedList *nlRef, QueryProcessor *qpRef)
{
    nl = nlRef;
    qp = qpRef;
    return (&mtreeAlg);
}
```

January-March 2008, Mirko Dibbert

7 Implementation file `MTree.cpp`

January-February 2008, Mirko Dibbert

7.1 Overview

This file contains the implementation of the `MTree` class.

```
#include <stack>
#include "MTree.h"

using namespace mtreeAlgebra;
```

Function *nearlyEqual*:

This auxiliary function is used in the search methods of the mtree class and returns true, if both numbers are `infiniy` or nearly equal.

```
template <typename FloatType>
inline bool nearlyEqual(FloatType a, FloatType b)
{
    FloatType infinity = numeric_limits<FloatType>::infinity();
    if (a == infinity)
        return (b == infinity);
    else if (b == infinity)
```

```
        return false;

    const FloatType scale = max(fabs(a), fabs(b));
    return  fabs(a - b) <=
            scale * 3 * numeric_limits<FloatType>::epsilon();
}
```

Default Constructor:

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

Constructor (load m-tree):

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

Copy constructor:

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

Method *registerNodePrototypes*:

```
void
MTree::registerNodePrototypes()
{
    // add internal node prototype
    addNodePrototype(new InternalNode(
        new NodeConfig(config.internalNodeConfig)));

    // add leaf node prototype
    addNodePrototype(new LeafNode(
        new NodeConfig(config.leafNodeConfig)));
}
```

Method *initialize*:

```
void
MTree::initialize()
{
    // init DistfunInfo object
    df_info = DistfunReg::getInfo(header.distfunName, header.dataId);

    // init MTreeConfig object
    config = MTreeConfigReg::getConfig(header.configName);

    // init Splitpol object
    splitpol = new Splitpol(
        config.promoteFun, config.partitionFun, df_info.distfun());

    if (nodeCacheEnabled)
        treeMngr->enableCache();
}
```

Method *initialize* :

```
void
MTree::initialize(DistDataId dataId, const string& distfunName,
                  const string& configName)
{
    if (isInitialized())
        return;

    // copy values to header
    header.dataId = dataId;
    strcpy(header.distfunName, distfunName.c_str());
    strcpy(header.configName, configName.c_str());

    initialize();
    header.initialized = true;

    registerNodePrototypes();

    //create root node
    NodePtr root(createLeaf(Leaf));
    header.root = root->getNodeId();
    ++header.leafCount;
    ++header.height;
}
```

Method *split* :

```
void MTree::split()
{
    bool done = false;
    while (!done)
    {
        // create new node on current level
        NodePtr newNode(createNeighbourNode(
            treeMngr->curNode()->isLeaf() ? Leaf : Internal));
```

```
// update node count, split node
if (treeMngr->curNode()->isLeaf())
{
    ++header.leafCount;
    splitpol->apply(treeMngr->curNode(), newNode, true);
}
else
{
    ++header.internalCount;
    splitpol->apply(treeMngr->curNode(), newNode, false);
}

#ifdef MTREE_PRINT_SPLIT_INFO
cmsg.info() << "\nsplit: splitted nodes contain "
            << treeMngr->curNode()->entryCount() << " / "
            << newNode->entryCount() << " entries." << endl;
cmsg.send();
#endif

// set modified flag to true and recompute node size
treeMngr->recomputeSize(treeMngr->curNode());
treeMngr->recomputeSize(newNode);

// retrieve promote entries
InternalEntry* promL = splitpol->getPromL();
InternalEntry* promR = splitpol->getPromR();

// insert promoted entries into routing nodes
if (treeMngr->hasParent())
{
    treeMngr->replaceParentEntry(promL);

    // insert promR
    if (!treeMngr->insert(treeMngr->parentNode(), promR))
        done = true;

    treeMngr->getParent();
    if (treeMngr->hasParent())
    {
        // update dist from promoted entries to their parents
        DFUN_RESULT distL, distR;
        DistData* data =
            treeMngr->parentEntry<InternalNode>()->data();
        df_info.dist(promL->data(), data, distL);
        df_info.dist(promR->data(), data, distR);
        promL->setDist(distL);
        promR->setDist(distR);
    }
}
else
{   // insert new root
    NodePtr newRoot(createRoot(Internal));
```

```
            ++header.height;
            ++header.internalCount;
            treeMngr->insert(newRoot, promL);
            treeMngr->insert(newRoot, promR);
            header.root = newRoot->getNodeId();
            done = true;
        }
    } // while
}
```

Method *intert* (`Attribute` objects):

```
void
MTree::insert(Attribute* attr, TupleId tupleId)
{
    // create new leaf entry
    LeafEntry* entry;
    try
    {
        entry = new LeafEntry(tupleId, df_info.getData(attr));
    }
    catch (bad_alloc&)
    {
        cmsg.warning() << "Not enough memory to create new entry, "
                       << "disabling node cache... "
                       << endl;
        cmsg.send();
        treeMngr->disableCache();

        try
        {
            entry = new LeafEntry(tupleId, df_info.getData(attr));
        }
        catch (bad_alloc&)
        {
            cmsg.error() << "Not enough memory to create new entry!"
                         << endl;
            cmsg.send();
        }
    }
    insert(entry, tupleId);
}
```

Method *insert* (`DistData` objects):

```
void
MTree::insert(DistData* data, TupleId tupleId)
{
    // create new leaf entry
    LeafEntry* entry;
    try
    {
        entry = new LeafEntry(tupleId, data);
```

```
    }
    catch (bad_alloc&)
    {
        cmsg.warning() << "Not enough memory to create new entry, "
                       << "disabling node cache... "
                       << endl;
        cmsg.send();
        treeMngr->disableCache();

        try
        {
            entry = new LeafEntry(tupleId, data);
        }
        catch (bad_alloc&)
        {
            cmsg.error() << "Not enough memory to create new entry!"
                         << endl;
            cmsg.send();
        }
    }
    insert(entry, tupleId);
}
```

Method *insert* (`LeafEntry` objects):

```
void MTree::insert(LeafEntry* entry, TupleId tupleId)
{
    #ifdef MTREE_DEBUG
    assert(isInitialized());
    #endif

    #ifdef MTREE_PRINT_INSERT_INFO
    if ((header.entryCount % insertInfoInterval) == 0)
    {
        const string clearline = "\r" + string(70, ' ') + "\r";
        cmsg.info() << clearline
                    << "entries: " << header.entryCount
                    << ", routing/leaf nodes: "
                    << header.internalCount << "/"
                    << header.leafCount;
        if(nodeCacheEnabled)
        {
            cmsg.info() << ", cache used: "
                        << treeMngr->cacheSize()/1024 << " kb";
        }
        cmsg.send();
    }
    #endif

    // init path
    treeMngr->initPath(header.root, header.height-1);

    // descent tree until leaf level
```

```
while (!treeMngr->curNode()->isLeaf())
{ /* find best path (follow the entry with the nearest dist to
     new entry or the smallest covering radius increase) */
    list<SearchBestPathEntry> entriesIn;
    list<SearchBestPathEntry> entriesOut;

    InternalNodePtr node =
        treeMngr->curNode()->cast<InternalNode>();

    for(unsigned i=0; i<node->entryCount(); ++i)
    {
        DFUN_RESULT dist;
        df_info.dist(node->entry(i)->data(), entry->data(), dist);
        if (dist <= node->entry(i)->rad())
        {
            entriesIn.push_back(
                SearchBestPathEntry(node->entry(i), dist, i));
        }
        else
        {
            entriesOut.push_back(
                SearchBestPathEntry(node->entry(i), dist, i));
        }
    }
    list<SearchBestPathEntry>::iterator best;

    if (!entriesIn.empty())
    { // select entry with nearest dist to new entry
        // (covering radius must not be increased)
        best = entriesIn.begin();
        list<SearchBestPathEntry>::iterator it;
        for (it = entriesIn.begin(); it != entriesIn.end(); ++it)
        {
            if (it->dist < best->dist)
                best = it;
        }
    }
    else
    { // select entry with minimal radius increase
        DFUN_RESULT dist;
        df_info.dist(entriesOut.front().entry->data(),
                entry->data(), dist);
        DFUN_RESULT minIncrease =
                dist - entriesOut.front().entry->rad();
        DFUN_RESULT minDist = dist;

        best = entriesOut.begin();
        list<SearchBestPathEntry>::iterator it;
        for (it = entriesIn.begin(); it != entriesIn.end(); ++it)
        {
            df_info.dist(it->entry->data(), entry->data(), dist);
            DFUN_RESULT increase = dist - it->entry->rad();
            if (increase < minIncrease)
```

```
                {
                    minIncrease = increase;
                    best = it;
                    minDist = dist;
                }
            }

            // update increased covering radius
            best->entry->setRad(minDist);
            node->setModified();
        }
        treeMngr->getChield(best->index);
    }

    //   compute distance from entry to parent node, if exist
    if (treeMngr->hasParent())
    {
        DFUN_RESULT dist;
        df_info.dist(entry->data(),
                treeMngr->parentEntry<InternalNode>()->data(), dist);
        entry->setDist(dist);
    }

    // insert entry into leaf, split if neccesary
    if (treeMngr->insert(treeMngr->curNode(), entry))
        split();

    ++header.entryCount;
}
```

Method *rangeSearch* :

```
void MTree::rangeSearch(DistData* data,
                        const DFUN_RESULT& searchRad,
                        list<TupleId>* results)
{
  #ifdef MTREE_DEBUG
  assert(isInitialized());
  #endif
    cout << treeMngr->cacheSize()/1024 << " kb, open nodes: "
         << openNodes() << "/" << openEntries() << "\t";

  results->clear();
  list< pair<DFUN_RESULT, TupleId> > resultList;

  stack<RemainingNodesEntry> remainingNodes;
  remainingNodes.push(RemainingNodesEntry(header.root, 0));

  #ifdef MTREE_PRINT_SEARCH_INFO
  unsigned entryCount = 0;
  unsigned nodeCount = 0;
  unsigned distComputations = 0;
  #endif
```

```
NodePtr node;

while(!remainingNodes.empty())
{
  #ifdef MTREE_PRINT_SEARCH_INFO
  nodeCount++;
  #endif

  node = getNode(remainingNodes.top().nodeId);
  DFUN_RESULT distQueryParent = remainingNodes.top().dist;
  remainingNodes.pop();

  if(node->isLeaf())
  {
    LeafNodePtr curNode = node->cast<LeafNode>();
    for(LeafNode::iterator it = curNode->begin();
        it != curNode->end(); ++it)
    {
      DFUN_RESULT dist = (*it)->dist();
      DFUN_RESULT distDiff = fabs(distQueryParent - dist);
      if ((distDiff  < searchRad) ||
           nearlyEqual<DFUN_RESULT>(distDiff, searchRad))
      {
        #ifdef MTREE_PRINT_SEARCH_INFO
        entryCount++;
        distComputations++;
        #endif

        DFUN_RESULT distQueryCurrent;
        df_info.dist(data, (*it)->data(), distQueryCurrent);
        if ((distQueryCurrent < searchRad) ||
            nearlyEqual<DFUN_RESULT>(distQueryCurrent, searchRad))
        {
          resultList.push_back(pair<DFUN_RESULT, TupleId>(
              distQueryCurrent, (*it)->tid()));
        }
      } // if
    } // for
  } else
  {
    InternalNodePtr curNode = node->cast<InternalNode>();
    for(InternalNode::iterator it = curNode->begin();
        it != curNode->end(); ++it)
    {
      DFUN_RESULT dist = (*it)->dist();
      DFUN_RESULT radSum = searchRad + (*it)->rad();
      DFUN_RESULT distDiff = fabs(distQueryParent - dist);
      if ((distDiff  < radSum) ||
           nearlyEqual<DFUN_RESULT>(distDiff, radSum))
      {
        #ifdef MTREE_PRINT_SEARCH_INFO
        distComputations++;
```

```
          #endif

          DFUN_RESULT newDistQueryParent;
          df_info.dist(data, (*it)->data(), newDistQueryParent);
          if ((newDistQueryParent < radSum) ||
              nearlyEqual<DFUN_RESULT>(newDistQueryParent, radSum))
          {
            remainingNodes.push(RemainingNodesEntry(
                (*it)->chield(), newDistQueryParent));
          }
        } // if
      } // for
    } // else
  } // while

  delete data;

  resultList.sort();
  list<pair<DFUN_RESULT, TupleId> >::iterator it = resultList.begin();
  while (it != resultList.end())
  {
    results->push_back(it->second);
    it++;
  }

  #ifdef MTREE_PRINT_SEARCH_INFO
  unsigned maxNodes = header.internalCount + header.leafCount;
  unsigned maxEntries = header.entryCount;
  unsigned maxDistComputations = maxNodes + maxEntries - 1;
  cmsg.info()
      << "Distance computations : " << distComputations << "\t(max "
      << maxDistComputations << ")" << endl
      << "Nodes analyzed        : " << nodeCount << "\t(max "
      << maxNodes << ")" << endl
      << "Entries analyzed      : " << entryCount << "\t(max "
      << maxEntries << ")" << endl << endl;
  cmsg.send();
  #endif
}
```

Method *nnSearch* :

```
void MTree::nnSearch(DistData* data, int nncount,
                          list<TupleId>* results)
{
  #ifdef MTREE_DEBUG
  assert(isInitialized());
  #endif

  results->clear();

  // init nearest neighbours array
  list< NNEntry > nearestNeighbours;
```

```
for (int i=0; i<nncount; i++)
{
  nearestNeighbours.push_back(
      NNEntry(0, numeric_limits<DFUN_RESULT>::infinity()));
}

vector< RemainingNodesEntryNNS > remainingNodes;

#ifdef MTREE_PRINT_SEARCH_INFO
unsigned entryCount = 0;
unsigned nodeCount = 0;
unsigned distComputations = 0;
#endif

remainingNodes.push_back(
    RemainingNodesEntryNNS(header.root, 0, 0));


while(!remainingNodes.empty())
{
  #ifdef MTREE_PRINT_SEARCH_INFO
  nodeCount++;
  #endif

  // read node with smallest minDist
  NodePtr node = getNode(remainingNodes.front().nodeId);
  DFUN_RESULT distQueryParent =
          remainingNodes.front().distQueryParent;
  DFUN_RESULT searchRad = nearestNeighbours.back().dist;

  // remove entry from remainingNodes heap
  pop_heap(remainingNodes.begin(), remainingNodes.end(),
            greater< RemainingNodesEntryNNS >());
  remainingNodes.pop_back();

  if (node->isLeaf())
  {
    LeafNodePtr curNode = node->cast<LeafNode>();
    for(LeafNode::iterator it = curNode->begin();
                           it != curNode->end(); ++it)
    {
      DFUN_RESULT distDiff = fabs(distQueryParent - (*it)->dist());
      if ((distDiff < searchRad) ||
           nearlyEqual<DFUN_RESULT>(distDiff, searchRad))
      {
        #ifdef MTREE_PRINT_SEARCH_INFO
        entryCount++;
        distComputations++;
        #endif

        DFUN_RESULT distQueryCurrent;
        df_info.dist(data, (*it)->data(), distQueryCurrent);
```

```
if ((distQueryCurrent < searchRad) ||
     nearlyEqual<DFUN_RESULT>(distQueryCurrent, searchRad))
{

  list<NNEntry>::iterator nnIter;
  nnIter = nearestNeighbours.begin();

  while ((distQueryCurrent > nnIter->dist) &&
          (nnIter != nearestNeighbours.end()))
  {
    nnIter++;
  }

  bool done = false;
  if (nnIter != nearestNeighbours.end())
  {
    TupleId tid = (*it)->tid();
    DFUN_RESULT dist = distQueryCurrent;

    while (!done && (nnIter != nearestNeighbours.end()))
    {
      if (nnIter->tid == 0)
      {
        nnIter->dist = dist;
        nnIter->tid = tid;
        done = true;
      }
      else
      {
        swap(dist, nnIter->dist);
        swap(tid, nnIter->tid);
      }
      nnIter++;
    }
  }

  searchRad = nearestNeighbours.back().dist;

  vector<RemainingNodesEntryNNS>::iterator
      it = remainingNodes.begin();

  while (it != remainingNodes.end())
  {
    if ((*it).minDist > searchRad)
    {
      swap(*it, remainingNodes.back());
      remainingNodes.pop_back();
    }
    else
      it++;
  }
  make_heap(remainingNodes.begin(),
             remainingNodes.end(),
```

```
                  greater<RemainingNodesEntryNNS>());

      } // if
    } // if
  } // for
} // if
else
{
  InternalNodePtr curNode = node->cast<InternalNode>();
  for(InternalNode::iterator it = curNode->begin();
      it != curNode->end(); ++it)
  {
    DFUN_RESULT distDiff = fabs(distQueryParent - (*it)->dist());
    DFUN_RESULT radSum = searchRad + (*it)->rad();
    if ((distDiff < radSum) ||
         nearlyEqual<DFUN_RESULT>(distDiff, radSum))
    {
      #ifdef MTREE_PRINT_SEARCH_INFO
      distComputations++;
      #endif

      DFUN_RESULT newDistQueryParent;
      df_info.dist(data, (*it)->data(), newDistQueryParent);

      DFUN_RESULT minDist, maxDist;
      minDist = max(newDistQueryParent - (*it)->rad(),
                    static_cast<DFUN_RESULT>(0));
      maxDist = newDistQueryParent + (*it)->rad();

      if ((minDist < searchRad) ||
           nearlyEqual<DFUN_RESULT>(minDist, searchRad))
      {
        // insert new entry into remainingNodes heap
        remainingNodes.push_back(RemainingNodesEntryNNS(
            (*it)->chield(), newDistQueryParent, minDist));
        push_heap(remainingNodes.begin(), remainingNodes.end(),
            greater<RemainingNodesEntryNNS>());

        if (maxDist < searchRad)
        {
          // update nearesNeighbours
          list<NNEntry>::iterator nnIter;
          nnIter = nearestNeighbours.begin();

          while ((maxDist > (*nnIter).dist) &&
                  (nnIter != nearestNeighbours.end()))
          {
            nnIter++;
          }

          if (((*nnIter).tid == 0) &&
               (nnIter != nearestNeighbours.end()))
          {
```

```
              if (!nearlyEqual<DFUN_RESULT>(
                      maxDist, (*nnIter).dist))
              {
                nearestNeighbours.insert(
                    nnIter, NNEntry(0, maxDist));
                nearestNeighbours.pop_back();
              }
            }

            searchRad = nearestNeighbours.back().dist;

            vector<RemainingNodesEntryNNS>::iterator it =
                remainingNodes.begin();

            while (it != remainingNodes.end())
            {
              if ((*it).minDist > searchRad)
              {
                it = remainingNodes.erase(it);
              }
              else
                it++;
            }
            make_heap(remainingNodes.begin(),
                      remainingNodes.end(),
                      greater<RemainingNodesEntryNNS>());
          }
        }
      }
    }
  }
} // while

delete data;
list< NNEntry >::iterator it;
for (it = nearestNeighbours.begin();
      it != nearestNeighbours.end(); it++)
{
  if ((*it).tid != 0)
  {
    results->push_back((*it).tid);
  }
}

#ifdef MTREE_PRINT_SEARCH_INFO
unsigned maxNodes = header.internalCount + header.leafCount;
unsigned maxEntries = header.entryCount;
unsigned maxDistComputations = maxNodes + maxEntries - 1;
cmsg.info()
    << "Distance computations : " << distComputations << "\t(max "
    << maxDistComputations << ")" << endl
    << "Nodes analyzed        : " << nodeCount << "\t(max "
    << maxNodes << ")" << endl
```

```
      << "Entries analyzed      : " << entryCount << "\t(max "
      << maxEntries << ")" << endl << endl;
  cmsg.send();
  #endif
}
```

```
This file is part of SECONDO.

Copyright (C) 2004, University in Hagen, Department of Computer Science,
Database Systems for New Applications.

SECONDO is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

SECONDO is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with SECONDO; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

January-February 2008, Mirko Dibbert

8 Implementation file `MTreeConfig.cpp`

This file implements the `MTreeConfig` class.

```
#include <limits>
#include "MTreeConfig.h"

using namespace mtreeAlgebra;
```

Initialize static members :

```
bool MTreeConfigReg::initialized = false;
map<string, MTreeConfig> MTreeConfigReg::configs;
string MTreeConfigReg::defaultConfigName = "undef";
```

Method *getConfig* :

```
MTreeConfig
MTreeConfigReg::getConfig(const string& name)
{
  if (!initialized)
    initialize();

  map< string, MTreeConfig >::iterator pos =
      configs.find(name);
```

```
  if (pos != configs.end())
    return pos->second;
  else
    return MTreeConfig();
}
```

Method *isDefined* :

```
bool
MTreeConfigReg::isDefined(const string& name)
{
  if (!initialized)
    initialize();

  map< string, MTreeConfig >::iterator pos =
      configs.find(name);

  if (pos != configs.end())
    return true;
  else
    return false;
}
```

Method *initialize* :

```
void
MTreeConfigReg::initialize()
{
```

Create node configs

```
  NodeConfig defaultleafConfig(
        Leaf, leafPrio, minLeafEntries, maxLeafEntries,
        minLeafPages, maxLeafPages, leafCacheable);

  NodeConfig defaultinternalConfig(
        Internal, internalPrio, minIntEntries, maxIntEntries,
        minIntPages, maxIntPages, internalCacheable);

  NodeConfig leafConfig40(
        Leaf, leafPrio, minLeafEntries, 40,
        minLeafPages, maxLeafPages, leafCacheable);

  NodeConfig internalConfig40(
        Internal, internalPrio, minIntEntries, 40,
        minIntPages, maxIntPages, internalCacheable);

  NodeConfig leafConfig80(
        Leaf, leafPrio, minLeafEntries, 80,
        minLeafPages, maxLeafPages, leafCacheable);
```

```
NodeConfig internalConfig80(
      Internal, internalPrio, minIntEntries, 80,
      minIntPages, maxIntPages, internalCacheable);
```

Set default config

```
defaultConfigName = "mlbdistHP";
```

Add config objects with unlimited entries per node.

```
configs["randomBal"] =  MTreeConfig(
    defaultleafConfig, defaultinternalConfig,
    RANDOM, BALANCED);

configs["mradBal"] =  MTreeConfig(
    defaultleafConfig, defaultinternalConfig,
    m_RAD, BALANCED);

configs["mmradBal"] =  MTreeConfig(
    defaultleafConfig, defaultinternalConfig,
    mM_RAD, BALANCED);

configs["mlbBal"] =  MTreeConfig(
    defaultleafConfig, defaultinternalConfig,
    M_LB_DIST, BALANCED);

configs["randomHP"] =  MTreeConfig(
    defaultleafConfig, defaultinternalConfig,
    RANDOM, GENERALIZED_HYPERPLANE);

configs["mradHP"] =  MTreeConfig(
    defaultleafConfig, defaultinternalConfig,
    m_RAD, GENERALIZED_HYPERPLANE);

configs["mmradHP"] =  MTreeConfig(
    defaultleafConfig, defaultinternalConfig,
    mM_RAD, GENERALIZED_HYPERPLANE);

configs["mlbdistHP"] =  MTreeConfig(
    defaultleafConfig, defaultinternalConfig,
    M_LB_DIST, GENERALIZED_HYPERPLANE);
```

Add config objects with max. 80 entries per node.

```
configs["randomBal80"] =  MTreeConfig(
    leafConfig80, internalConfig80,
    RANDOM, BALANCED);

configs["mradBal80"] =  MTreeConfig(
    leafConfig80, internalConfig80,
    m_RAD, BALANCED);
```

```
configs["mmradBal80"] =  MTreeConfig(
    leafConfig80, internalConfig80,
    mM_RAD, BALANCED);

configs["mlbBal80"] =  MTreeConfig(
    leafConfig80, internalConfig80,
    M_LB_DIST, BALANCED);

configs["randomHP80"] =  MTreeConfig(
    leafConfig80, internalConfig80,
    RANDOM, GENERALIZED_HYPERPLANE);

configs["mradHP80"] =  MTreeConfig(
    leafConfig80, internalConfig80,
    m_RAD, GENERALIZED_HYPERPLANE);

configs["mmradHP80"] =  MTreeConfig(
    leafConfig80, internalConfig80,
    mM_RAD, GENERALIZED_HYPERPLANE);

configs["mlbdistHP80"] =  MTreeConfig(
    leafConfig80, internalConfig80,
    M_LB_DIST, GENERALIZED_HYPERPLANE);
```

Add config objects with max. 40 entries per node.

```
configs["randomBal40"] =  MTreeConfig(
    leafConfig40, internalConfig40,
    RANDOM, BALANCED);

configs["mradBal40"] =  MTreeConfig(
    leafConfig40, internalConfig40,
    m_RAD, BALANCED);

configs["mmradBal40"] =  MTreeConfig(
    leafConfig40, internalConfig40,
    mM_RAD, BALANCED);

configs["mlbBal40"] =  MTreeConfig(
    leafConfig40, internalConfig40,
    M_LB_DIST, BALANCED);

configs["randomHP40"] =  MTreeConfig(
    leafConfig40, internalConfig40,
    RANDOM, GENERALIZED_HYPERPLANE);

configs["mradHP40"] =  MTreeConfig(
    leafConfig40, internalConfig40,
    m_RAD, GENERALIZED_HYPERPLANE);

configs["mmradHP40"] =  MTreeConfig(
    leafConfig40, internalConfig40,
    mM_RAD, GENERALIZED_HYPERPLANE);
```

```
  configs["mlbdistHP40"] =  MTreeConfig(
      leafConfig40, internalConfig40,
      M_LB_DIST, GENERALIZED_HYPERPLANE);

  initialized = true;
}
```