M-Tree Algebra

Contents

1 MTreeAlgebra

November 2007, Mirko Dibbert

1.1 Overview

TODO insert algebra description

```
#ifndef __MTREE_ALGEBRA_H
#define __MTREE_ALGEBRA_H

// #define __MT_DEBUG
// #define __MT_PRINT_ENTRY_INFO
// #define __MT_PRINT_NODE_INFO

#define __MT_PRINT_NODE_CACHE_INFO
#define __MT_PRINT_CONFIG_INFO
// #define __MT_PRINT_SPLIT_INFO
#define __MT_PRINT_INSERT_INFO
#define __MT_PRINT_SEARCH_INFO

#include <stack>
#include "StandardTypes.h"
#include "WinUnix.h"
#include "LogMsg.h"
#include "StandardTypes.h"
#include "RelationAlgebra.h"
```

```
#include "MetricRegistry.h"
#include "MetricalAttribute.h"

using namespace std;

namespace MT
{

const unsigned NODE_PAGESIZE = ( WinUnix::getPageSize() - 60 );
```

Size of a m-tree node. If an error like

```
DbEnv: Record size of x too large for page size of y
```

---

occurs, the integer value needs to be increased!

```
const unsigned MAX_CACHED_NODES = 1024;
```

The maximum number of nodes, which should be hold open in the node cache

```
const bool ROOT = true;
const bool SIZE_CHANGED = true;
```

Some constants to make the source better readable.

```
}

#endif
```

```
along with SECONDO; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

November 2007, Mirko Dibbert

```
using namespace std;

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

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

1.2 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 = ten "
                          "createmtree " ) ) ) );
}

ListExpr
OutMTree( ListExpr typeInfo, Word  value )
{
  return nl->OneElemList( nl->StringAtom(
  "should output some statistic infos in final vesion" ));
}

Word
InMTree( ListExpr typeInfo, ListExpr value,
        int errorPos, ListExpr &errorInfo, bool &correct )
{
  correct = false;
  return SetWord( 0 );
}
```

```
ListExpr
SaveToListMTree( ListExpr typeInfo, Word  value )
{
  return nl->IntAtom( 0 );
}

Word
RestoreFromListMTree( ListExpr typeInfo, ListExpr value,
               int errorPos, ListExpr &errorInfo,
               bool &correct )
{
  return SetWord( Address( 0 ) );
}

Word
CreateMTree( const ListExpr typeInfo )
{
  return SetWord( new MT::MTree() );
}

void
DeleteMTree( const ListExpr typeInfo, Word &w )
{
  MT::MTree *mtree = ( MT::MTree* )w.addr;
  mtree->deleteFile();
  delete mtree;
}

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

  MT::MTree* mtree = new MT::MTree( fileid );
  value = SetWord( mtree );
  return true;
}

bool
SaveMTree( SmiRecord &valueRecord,
       size_t &offset,
       const ListExpr typeInfo,
       Word &value )
{
  SmiFileId fileId;
  MT::MTree *mtree = ( MT::MTree* )value.addr;
  fileId = mtree->getFileId();
  if (fileId)
```

```
  {
    valueRecord.Write( &fileId, sizeof( SmiFileId ), offset );
    offset += sizeof( SmiFileId );
    return true;
  }
  else
  {
    return false;
  }
}

void CloseMTree( const ListExpr typeInfo, Word &w )
{
  MT::MTree *mtree = ( MT::MTree* )w.addr;
  delete mtree;
}

Word CloneMTree( const ListExpr typeInfo, const Word &w )
{
  return SetWord( 0 );
}

void *CastMTree( void *addr )
{
  return ( 0 );
}

int SizeOfMTree()
{
  return 0;
}

bool CheckMTree( ListExpr type, ListExpr &errorInfo )
{
  //TODO not yet implemented
  return true;
}

TypeConstructor
mtree( "mtree",      MTreeProp,
     OutMTree,    InMTree,
     SaveToListMTree, RestoreFromListMTree,
     CreateMTree,   DeleteMTree,
     OpenMTree,     SaveMTree,
     CloseMTree,    CloneMTree,
     CastMTree,     SizeOfMTree,
     CheckMTree );
```

1.3 Operators

1.3.1 Operator *createmtree*

```
int
CreateMTreeValueMapping_Rel( Word  *args, Word  &result,
               int message, Word  &local, Supplier s )
{
  result = qp->ResultStorage( s );
  MT::MTree *mtree = ( MT::MTree* )result.addr;

  Relation* relation = ( Relation* )args[0].addr;
  int attrIndex = (( CcInt* )args[4].addr )->GetIntval();
  string mfName = (( CcString* )args[5].addr )->GetValue();
  string configName = (( CcString* )args[6].addr )->GetValue();
  Tuple *tuple;
  cout << mfName << " " << configName << endl;
  GenericRelationIterator *iter = relation->MakeScan();
  while (( tuple = iter->GetNextTuple() ) != 0 )
  {
    AttributeType type =
        tuple->GetTupleType()-> GetAttributeType( attrIndex );

    Attribute* attr = tuple->GetAttribute( attrIndex );
    if( attr->IsDefined() )
    {
      if ( !mtree->isInitialized() )
      {
        mtree->initialize( attr, am->Constrs(
            type.algId, type.typeId ), mfName, configName );
      }
      mtree->insert( attr, tuple->GetTupleId() );
    }
    tuple->DeleteIfAllowed();
  }
  delete iter;

  #ifdef MT_PRINT_INSERT_INFO
  cmsg.info() << endl;
  cmsg.send();
  #endif
//   mtree->print();
//         try
//         {
//           //new ...
//         }
//         catch (bad_alloc&)
//         {
//         }

  return 0;
}
```

```
int CreateMTreeValueMapping_Stream( Word  *args, Word  &result,
                  int message, Word  &local, Supplier s )
{
  result = qp->ResultStorage( s );
  MT::MTree *mtree = ( MT::MTree* )result.addr;

  int attrIndex = (( CcInt* )args[4].addr )->GetIntval();
  string mfName = (( CcString* )args[5].addr )->GetValue();
  string configName = (( CcString* )args[6].addr )->GetValue();

  Word wTuple;

  assert(mtree != 0);

  qp->Open(args[0].addr);
  qp->Request(args[0].addr, wTuple);
  while (qp->Received(args[0].addr))
  {
    Tuple* tuple = (Tuple*)wTuple.addr;
    Attribute* attr = tuple->GetAttribute( attrIndex );
    AttributeType type =
        tuple->GetTupleType()->GetAttributeType( attrIndex );
    if( attr->IsDefined() )
    {
      if ( !mtree->isInitialized() )
      {
        mtree->initialize( attr, am->Constrs(
            type.algId, type.typeId ), mfName, configName );
      }
      mtree->insert( attr, tuple->GetTupleId() );
    }
    tuple->DeleteIfAllowed();
    qp->Request(args[0].addr, wTuple);
  }
  qp->Close(args[0].addr);

  return 0;
}


int CreateMTreeSelect( ListExpr args )
{
  if ( nl->IsEqual( nl->First( nl->First( args ) ), "rel" ) )
    return 0;

  if ( nl->IsEqual( nl->First( nl->First( args ) ), "stream" ) )
    return 1;

  return -1;
}

ValueMapping CreateMTreeMap[] = { CreateMTreeValueMapping_Rel,
                  CreateMTreeValueMapping_Stream
```

```
                  };

ListExpr CreateMTreeTypeMapping( ListExpr args )
{
  string errmsg;
  bool cond;
  NList nl_args( args );

  errmsg = "Operator createmtree expects three arguments.";
  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 relation or stream)
  cond = !(arg1.isAtom()) &&
         (
           ( arg1.first().isEqual( "rel" ) &&
             IsRelDescription( arg1.listExpr() )) ||
           ( arg1.first().isEqual( "stream" ) &&
             IsStreamDescription( arg1.listExpr() ))
         );
  errmsg = "Operator createmtree expects a list with structure\n"
           "   rel (tuple ((a1 t1)...(an tn))) or\n"
           "   stream (tuple ((a1 t1)...(an tn)))\n"
           "as first argument, but got a list with structure '" +
       arg1.convertToString() + "'.";
  CHECK_COND( cond , errmsg);

  // check, if third argument is an attribute name
  errmsg = "Operator createmtree expects an attribute name "
           "as fourth argument, but got '" +
           arg4.convertToString() + "'.";
  CHECK_COND( arg4.isSymbol(), errmsg);

  string attrName = arg4.str();
  NList tupleDescription = arg1.second();
  NList attrList = tupleDescription.second();

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

  // check, if attribute type is string, int, real or METRICAL
  ListExpr errorInfo = nl->OneElemList( nl->SymbolAtom( "ERRORS" ) );
```

```
  cond = attrType.isEqual( "string" ) ||
         attrType.isEqual( "int" ) ||
         attrType.isEqual( "real" ) ||
         am->CheckKind( "METRICAL", attrType.listExpr(), errorInfo );
  errmsg = "Operator createmtree expects an attribute of type "
           "string, int, real or METRICAL as third argument, but got"
           " '" + attrType.convertToString() + "'.";
  CHECK_COND( cond, errmsg );

  // check if the metric given in second argument is defined
  errmsg = "Operator createmtree expects the name of a registered"
           "metric as second argument, but got a list with structure"
           " '" + arg2.convertToString() + "'.";
  CHECK_COND( arg2.isSymbol(), errmsg);
  string mfName = arg2.str();
  errmsg = "Metric " + mfName + " for type constructor " +
           attrType.convertToString() + " not defined!";
  cond = MetricRegistry::getMetric(
      attrType.convertToString(), mfName ) != 0;

  errmsg = "Operator createmtree expects the name of a registered"
           "mtree-config object as third argument, but got a list "
           "with structure '" + arg2.convertToString() + "'.";
  CHECK_COND( arg3.isSymbol(), errmsg);
  string configName = arg3.str();
  // TODO type checking for config name

  NList result (
      NList( "APPEND" ),
      NList(
        attrIndex - 1,
        NList ( mfName, true ),
        NList ( configName, true ) ),
      NList( NList( "mtree" ), tupleDescription, attrType ) );
  cout << result.convertToString() << endl;
  return result.listExpr();
}

struct CreateMTreeInfo : OperatorInfo
{
  CreateMTreeInfo()
  {
    name = "createmtree";
    signature = "rel x string x id -> mtree";
    syntax = "_ _ createmtree [ _ ]";
    meaning = "string should be the name of the metric.";
    example =
      "let mtree_index = Rel DEFAULT createmtree [key]";
    remark = "";
  }
};
```

1.4 Operator range

```
struct RangeSearchLocalInfo
{
  Relation* relation;
  list<TupleId>* results;
  list<TupleId>::iterator iter;

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

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

  ~RangeSearchLocalInfo()
  {
    delete results;
  }

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

int RangeSearchValueMapping_Rel( Word  *args, Word  &result,
                  int message, Word  &local, Supplier s )
{
  RangeSearchLocalInfo *localInfo;

  switch (message)
  {
    case OPEN :
    {
      localInfo = new RangeSearchLocalInfo(
          static_cast<Relation*>( args[0].addr ) );
      MT::MTree* mtree = static_cast<MT::MTree*>( args[1].addr );
      Attribute* attr = static_cast<Attribute*>( args[2].addr );
      double searchRad = ((CcReal*)args[3].addr)->GetValue();
```

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

      assert(localInfo->relation != 0);
      local = SetWord(localInfo);
      return 0;
    }

    case REQUEST :
    {
      localInfo = (RangeSearchLocalInfo*)local.addr;

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

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

int
RangeSearchValueMapping_Stream( Word  *args, Word  &result,
               int message, Word  &local, Supplier s )
{
  return 0;
}

int RangeSearchSelect( ListExpr args )
{
  if ( nl->IsEqual( nl->First( nl->First( args ) ), "rel" ) )
    return 0;

  if ( nl->IsEqual( nl->First( nl->First( args ) ), "stream" ) )
    return 1;

  return -1;
}
```

```
ValueMapping RangeSearchMap[] = { RangeSearchValueMapping_Rel,
                  RangeSearchValueMapping_Stream
                };


ListExpr RangeSearchTypeMapping( ListExpr args )
{
  string errmsg;
  bool cond;
  NList nl_args( args );

  errmsg = "Operator range expects three arguments.";
  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 relation or stream)
  cond = !(arg1.isAtom()) &&
         (
           ( arg1.first().isEqual( "rel" ) &&
             IsRelDescription( arg1.listExpr() )) ||
           ( arg1.first().isEqual( "stream" ) &&
             IsStreamDescription( arg1.listExpr() ))
         );
  errmsg = "Operator createmtree expects a list with structure\n"
           "   rel (tuple ((a1 t1)...(an tn))) or\n"
           "   stream (tuple ((a1 t1)...(an tn)))\n"
           "as first argument, but got a list with structure '" +
      arg1.convertToString() + "'.";
  CHECK_COND( cond , errmsg);

  // check second argument
  errmsg = "Operator rangesearch expects a mtree "
           "as second argument, but got '" +
           arg2.convertToString() + "'.";
  CHECK_COND( arg2.first().isEqual( "mtree" ), errmsg );

  // check third argument
  errmsg = "Operator createmtree expects an attribute of type "
           " string, int, real or METRICAL as third argument, but "
           "got '" + arg3.convertToString() + "'.";
  ListExpr errorInfo = nl->OneElemList( nl->SymbolAtom( "ERRORS" ) );
  cond = arg3.isEqual( "string" ) ||
         arg3.isEqual( "int" ) ||
         arg3.isEqual( "real" ) ||
         am->CheckKind( "METRICAL", arg3.listExpr(), errorInfo );
  CHECK_COND( cond, errmsg );

  // check if used attribute is equal to attribute used in m-tree
  cond = arg2.third().isEqual( arg3.convertToString() );
```

```
  errmsg = "The used m-tree contains attributes of type " +
           arg2.third().convertToString() + ", but the given "
           " attribute argument is of type " +
           arg3.convertToString();
  CHECK_COND( cond, errmsg );

  // check fourth argument
  errmsg = "Operator createmtree expects an real value as fourth "
           "argument, but got '" + arg4.convertToString() + "'.";
  CHECK_COND( arg4.isEqual( "real" ), errmsg );

  return
    nl->TwoElemList(
      nl->SymbolAtom("stream"),
      arg1.second().listExpr());
}

struct RangeSearchInfo : OperatorInfo
{
  RangeSearchInfo()
  {
    name = "rangesearch";
    signature = "m-tree x string x attribute x int -> rel";
    syntax = "_ range [ _, _, _ ]";
    meaning = "string should be the name of the metric.";
    example =
      "query mtree_index range [DEFAULT, queryattr, 2]";
    remark = "";
  }
};
```

1.5 Operator nnsearch

```
struct NNSearchLocalInfo
{
  Relation* relation;
  list<TupleId>* results;
  list<TupleId>::iterator iter;

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

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

  ~NNSearchLocalInfo()
  {
    delete results;
```

```
  }

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

int NNSearchValueMapping_Rel( Word  *args, Word  &result,
                  int message, Word  &local, Supplier s )
{
  NNSearchLocalInfo *localInfo;

  switch (message)
  {
    case OPEN :
    {
      localInfo = new NNSearchLocalInfo(
          static_cast<Relation*>( args[0].addr ) );
      MT::MTree* mtree = static_cast<MT::MTree*>( args[1].addr );
      Attribute* attr = static_cast<Attribute*>( args[2].addr );
      int nncount= ((CcInt*)args[3].addr)->GetValue();

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

      assert(localInfo->relation != 0);
      local = SetWord(localInfo);
      return 0;
    }

    case REQUEST :
    {
      localInfo = (NNSearchLocalInfo*)local.addr;

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

```
        return CANCEL;
      }
    }

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

int
NNSearchValueMapping_Stream( Word  *args, Word  &result,
               int message, Word  &local, Supplier s )
{
  return 0;
}

int NNSearchSelect( ListExpr args )
{
  if ( nl->IsEqual( nl->First( nl->First( args ) ), "rel" ) )
    return 0;

  if ( nl->IsEqual( nl->First( nl->First( args ) ), "stream" ) )
    return 1;

  return -1;
}

ValueMapping NNSearchMap[] = { NNSearchValueMapping_Rel,
                  NNSearchValueMapping_Stream
                 };


ListExpr NNSearchTypeMapping( ListExpr args )
{
  string errmsg;
  bool cond;
  NList nl_args( args );

  errmsg = "Operator nnsearch expects three arguments.";
  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 relation or stream)
  cond = !(arg1.isAtom()) &&
```

```
        (
          ( arg1.first().isEqual( "rel" ) &&
            IsRelDescription( arg1.listExpr() )) ||
          ( arg1.first().isEqual( "stream" ) &&
            IsStreamDescription( arg1.listExpr() ))
        );
  errmsg = "Operator nnsearch expects a list with structure\n"
           "   rel (tuple ((a1 t1)...(an tn))) or\n"
           "   stream (tuple ((a1 t1)...(an tn)))\n"
           "as first argument, but got a list with structure '" +
       arg1.convertToString() + "'.";
  CHECK_COND( cond , errmsg);

  // check second argument
  errmsg = "Operator nnearch expects a mtree "
           "as second argument, but got '" +
           arg2.convertToString() + "'.";
  CHECK_COND( arg2.first().isEqual( "mtree" ), errmsg );

  // check third argument
  errmsg = "Operator nnsearch expects an attribute of type "
           " string, int, real or METRICAL as third argument, but "
           "got '" + arg3.convertToString() + "'.";
  ListExpr errorInfo = nl->OneElemList( nl->SymbolAtom( "ERRORS" ) );
  cond = arg3.isEqual( "string" ) ||
         arg3.isEqual( "int" ) ||
         arg3.isEqual( "real" ) ||
         am->CheckKind( "METRICAL", arg3.listExpr(), errorInfo );
  CHECK_COND( cond, errmsg );

  // check if used attribute is equal to attribute used in m-tree
  cond = arg2.third().isEqual( arg3.convertToString() );
  errmsg = "The used m-tree contains attributes of type " +
           arg2.third().convertToString() + ", but the given "
           " attribute argument is of type " +
           arg3.convertToString();
  CHECK_COND( cond, errmsg );

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

  return
    nl->TwoElemList(
      nl->SymbolAtom("stream"),
      arg1.second().listExpr());
}

struct NNSearchInfo : OperatorInfo
{
  NNSearchInfo()
  {
```

```
    name = "nnsearch";
    signature = "";
    syntax = "";
    meaning = "";
    example = "";
    remark = "";
  }
};
```

1.6 Create and initialize the Algebra

```
class MTreeAlgebra : public Algebra
{

public:
  MTreeAlgebra() : Algebra()
  {
    AddTypeConstructor( &mtree );
    AddOperator(
        CreateMTreeInfo(),
        CreateMTreeMap,
        CreateMTreeSelect,
        CreateMTreeTypeMapping );

    AddOperator(
        RangeSearchInfo(),
        RangeSearchMap,
        RangeSearchSelect,
        RangeSearchTypeMapping );

    AddOperator(
        NNSearchInfo(),
        NNSearchMap,
        NNSearchSelect,
        NNSearchTypeMapping );
  }

  ~MTreeAlgebra() {};
};

MTreeAlgebra mtreeAlgebra;

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

2 The M-Tree Datastructure

December 2007, Mirko Dibbert

2.1 Overview

TODO enter datastructure description

2.2 Class *MTree*

2.2.1 Class description

TODO enter class description

2.2.2 Definition part (file: MTree.h)

```
#ifndef __MTREE_H
#define __MTREE_H

#include "MTNodeMngr.h"
#include "MTSplitpol.h"
#include "MTreeConfig.h"

namespace MT
{
```

```
class MTree
{
  struct Header
  {
    STRING_T tcName;     // type name of the stored entries
    STRING_T metricName; // name of the used metric
    STRING_T configName; // name of the MTreeConfig object
    SmiRecordId root;    // page of the root node
    unsigned height;
    unsigned entryCount;
    unsigned routingCount;
    unsigned leafCount;

    Header() :
      root  ( 0 ),
      height( 0 ),
      entryCount( 0 ),
      routingCount( 0 ),
      leafCount( 0 )
    {}
  }; // struct Header
```

This struct contains all neccesary data to reinitialize a previously stored m-tree.

```
  bool initialized;
  SmiRecordFile file;
  Header header;
  Splitpol* splitpol;
  NodeMngr* nodeMngr;

  TMetric metric;
  MTreeConfig config;
  vector<SmiRecordId> path;
  vector<unsigned> indizes;
  Node* nodePtr;

struct RemainingNodesEntry
{
  SmiRecordId nodeId;
  unsigned deepth;
  double dist;

  RemainingNodesEntry( SmiRecordId nodeId_, size_t deepth_,
                       double dist_ ) :
    nodeId( nodeId_ ),
    deepth( deepth_ ),
    dist  ( dist_ )
  {}

};
```

This struct is used in the `rangeSearch` method as path entry

```
struct SearchBestPathEntry
{
  SearchBestPathEntry( Entry* entry_, double dist_,
                       unsigned index_ ) :
    entry( entry_ ),
    dist( dist_ ),
    index( index_ )
  {}

  Entry* entry;
  double dist;
  unsigned index;
};
```

This struct is used in the `insert` method when searching for the best path to descent the tree.

```
  void readHeader();
```

Reads the header from file.

```
  void writeHeader();
```

Writes the header to file.

```
  void split(Entry* entry );
```

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

```
public:
  MTree();
```

Constructor, creates a new m-tree (`initialize` method must be called before the tree can be used).

```
  MTree( const SmiFileId fileid );
```

Constructor, opens an existing tree.

```
  ~MTree();
```

Destructor

```
  void initialize( const Attribute* attr,
                   const string tcName,
                   const string metricName,
                   const string configName );
```

This method initializes a new created m-tree.

```
  void deleteFile();
```

This Method deletes the m-tree file.

```
DistData* getDistData( Attribute* attr );
```

Returns a new DistData object which will be created from a CcInt, CcReal or CcString object or obtained from the getDistData method of attr.

```
inline SmiFileId getFileId()
{
  return file.GetFileId();
}
```

This method returns the file id of the `SmiRecordFile` containing the m-tree.

```
inline bool isInitialized()
{ return initialized; }
```

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

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

Inserts a new entry into the tree.

```
void rangeSearch( Attribute* attr, const double& searchRad,
                  list<TupleId>* results );
```

Returns all entries in the tree, wich have a maximum distance of `searchRad` to the attribute `attr` in the result list.

```
void nnSearch( Attribute* attr, int nncount,
               list<TupleId>* results );
```

k-nearest-neighbour search

```
}; // MTree

} // namespace MTree

#endif
```

/newpage

2.2.3 Implementation part (file: MTree.cpp)

```
#include "MTree.h"
```

Constructor (new m-tree):

```
MT::MTree::MTree()
: initialized( false ),
  file( true, NODE_PAGESIZE ),
  header(),
  splitpol( 0 ),
```

```
  nodeMngr( 0 )
{
  file.Create();

  // create header nodeId
  SmiRecordId headerId;
  SmiRecord headerRecord;
  file.AppendRecord( headerId, headerRecord );
  assert( headerId == 1 );
}
```

Constructor (load m-tree):

```
MT::MTree::MTree( const SmiFileId fileid )
: initialized( false ),
  file( true ),
  header(),
  splitpol( 0 ),
  nodeMngr( 0 )
{
  assert(file.Open( fileid ));
  readHeader();

  // get metric function
  metric = MetricRegistry::getMetric
      ( header.tcName, header.metricName );

  // get MTreeConfig object
  config = MTreeConfigReg::getMTreeConfig ( header.configName );

  // initialize node manager
  nodeMngr = new NodeMngr( &file, config.maxNodeEntries );

  // initialize split policy
  splitpol = new
      Splitpol( config.promoteFun, config.partitionFun, metric );

  initialized = true;
}
```

Destructor:

```
MT::MTree::~MTree()
{
  if ( file.IsOpen() )
  {
    writeHeader();
    file.Close();
  }

  delete splitpol;
  delete nodeMngr;
```

```
#ifdef __MT_DEBUG
  if ( Node::objectsOpen() )
  {
    cmsg.warning() << "*** Memory leak warning: "
                   << Node::objectsOpen()
                   << " <MT::Node> object(s) left open!" << endl;
    cmsg.send();
  }

  if ( Entry::objectsOpen() )
  {
    cmsg.warning() << "*** Memory leak warning: "
                   << Entry::objectsOpen()
                   << " <MT::Entry> object(s) left open!" << endl;
    cmsg.send();
  }
#endif
}
```

Method *deleteFile*

```
void
MT::MTree::deleteFile()
{
  if ( file.IsOpen() )
    file.Close();

  file.Drop();
}
```

Method *writeHeader* :

```
void
MT::MTree::writeHeader()
{
  SmiRecord record;
  file.SelectRecord( (SmiRecordId)1, record, SmiFile::Update );
  record.Write( &header, sizeof( Header ), 0 );
}
```

Method *readHeader* :

```
void MT::MTree::readHeader()
{
  SmiRecord record;
  file.SelectRecord( (SmiRecordId)1, record, SmiFile::ReadOnly );
  record.Read( &header, sizeof( Header ), 0 );
}
```

Method *initialize* :

```
void MT::MTree::initialize( const Attribute* attr,
                            const string tcName,
                            const string metricName,
                            const string configName )
{
  if ( initialized )
    return;

  // get metric function
  metric = MetricRegistry::getMetric ( tcName, metricName );

  // get MTreeConfig object
  config = MTreeConfigReg::getMTreeConfig( configName );

  // initialize node manager
  nodeMngr = new NodeMngr( &file, config.maxNodeEntries );

  // initialize split policy
  splitpol = new
      Splitpol( config.promoteFun, config.partitionFun, metric );

  //create root node
  Node* root = nodeMngr->createNode();
  header.leafCount++;
  header.height++;

  // update header
  strcpy( header.tcName, tcName.c_str() );
  strcpy( header.metricName, metricName.c_str() );
  strcpy( header.configName, configName.c_str() );
  header.root = root->getNodeId();

  root->deleteIfAllowed();

  initialized = true;
}
```

Method *getDistData* :

```
DistData*
MT::MTree::getDistData( Attribute* attr )
{
  DistData* data;
  string tcName ( header.tcName );
  if ( tcName == "int" )
  {
    int value = static_cast<CcInt*>(attr)->GetValue();
    char buffer[sizeof(int)];
    memcpy( buffer, &value, sizeof(int) );
    data = new DistData( sizeof(int), buffer );
  }
  else if  ( tcName == "real" )
  {
```

```
    SEC_STD_REAL value =
        static_cast<CcReal*>(attr)-> GetValue();
    char buffer[sizeof(SEC_STD_REAL)];
    memcpy( buffer, &value, sizeof(SEC_STD_REAL) );
    data = new DistData( sizeof(SEC_STD_REAL), buffer );
  }
  else if  ( tcName == "string" )
  {
    string value = static_cast<CcString*>( attr )-> GetValue();
    data = new DistData( value );
  }
  else
  {
    data = static_cast<MetricalAttribute*>( attr )->
        getDistData( header.metricName );
  }
  return data;
}
```

Method *split* :

```
void
MT::MTree::split( Entry* entry )
{
  unsigned char deepth = header.height - 1;

  while ( true )
  {
    bool isLeaf = ( deepth == (header.height-1) );

    // create new node
    Node* newNode;
    if ( isLeaf )
    {
      header.leafCount++;
      newNode = nodeMngr->createNode();
    }
    else
    {
      header.routingCount++;
      newNode = nodeMngr->createNode();
    }

    // get current entries, store new entry vector to node
    // (will be filled in splitpol->apply)
    vector<Entry*>* entries = new vector<Entry*>();
    nodePtr->swapEntries( entries );
    entries->push_back( entry );

    /* apply splitpol: this will split the entries given in the first
       vector to the second and third vector by using the promote and
       partition function defined in the current MTreeConfig object.
    */
```

```
splitpol->apply( entries, nodePtr->getEntries(),
                 newNode->getEntries(), isLeaf );
delete entries;

#ifdef __MT_PRINT_SPLIT_INFO
cmsg.info() << "\nsplit: splitted nodes contain "
            << nodePtr->getEntryCount() << " / "
            << newNode->getEntryCount() << " entries." << endl;
cmsg.send();
#endif

// set modified flag to true and recompute node size
nodePtr->modified( SIZE_CHANGED );
newNode->modified( SIZE_CHANGED );

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

// update chield pointers
promL->setChield( nodePtr->getNodeId() );
promR->setChield( newNode->getNodeId() );

newNode->deleteIfAllowed();

// insert new root
if ( deepth == 0 )
{
  header.routingCount++;
  Node* newRoot = nodeMngr->createNode();
  newRoot->insert( promL );
  newRoot->insert( promR );
  header.root = newRoot->getNodeId();
  header.height++;
  newRoot->deleteIfAllowed();
  return;
}
// insert promoted entries into routing nodes
else
{
  deepth--;
  nodePtr->deleteIfAllowed();
  nodePtr = nodeMngr->getNode( path[deepth] );

  // update distances to parent
  if ( deepth > 0)
  {
    double distL, distR;
    Node* parent = nodeMngr->getNode( path[deepth-1] );
    Entry* parentEntry =
        (*parent->getEntries())[indizes[deepth-1]];
    (*metric)( promL->data(), parentEntry->data(), distL );
    (*metric)( promR->data(), parentEntry->data(), distR );
```

```
        promL->setDist( distL );
        promR->setDist( distR );
        parent->deleteIfAllowed();
      }

      // replace old promoted entry with promL
      nodePtr->update( indizes[deepth], promL );

      // insert promR
      if (!nodePtr->insert( promR ))
        entry = promR;
      else
        return;
    } // else
  } // while
}
```

Method *insert* :

```
void
MT::MTree::insert( Attribute* attr, TupleId tupleId )
{
  #ifdef __MT_DEBUG
  assert( initialized );
  #endif

  #ifdef __MT_PRINT_INSERT_INFO
  if ((header.entryCount % 5000) == 0)
  {
    cmsg.info() << endl
                << "routing nodes: " << header.routingCount
                << "\tleaves: " << header.leafCount
                << "\theight: " << header.height
                << "\tentries: " << header.entryCount << "\t";
    cmsg.send();
  }
  else if ((header.entryCount % 100) == 0)
  {
    cmsg.info() << ".";
    cmsg.send();
  }
  #endif

  unsigned char deepth = 0;

  // init path vector
  path.clear();
  path.reserve( header.height + 1 );
  path.push_back( header.root );

  // init index vector
  indizes.clear();
  indizes.reserve( header.height );
```

```
// init node pointer
nodePtr = nodeMngr->getNode( header.root );

// create new entry
Entry* entry = new Entry( tupleId, getDistData(attr) );

// descent tree until leaf level
while ( deepth < header.height - 1 )
{ /* 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;

    vector<Entry*>* entries = nodePtr->getEntries();
    vector<Entry*>::iterator iter;
    unsigned index = 0;
    for ( iter = entries->begin();
          iter != entries->end();
          iter++, index++ )
    {
      double dist;
      (*metric)( (*iter)->data(), entry->data(), dist );
      if ( dist <= (*iter)->rad() )
      {
        entriesIn.push_back(
            SearchBestPathEntry( *iter, dist, index ) );
      }
      else
      {
        entriesOut.push_back(
            SearchBestPathEntry( *iter, dist, index ) );
      }
    } // for

    list<SearchBestPathEntry>::iterator best;
    if (!entriesIn.empty())
    { // select entry with nearest dist to new entry
      best = entriesIn.begin();
      list<SearchBestPathEntry>::iterator iter;
      for ( iter = entriesIn.begin();
            iter != entriesIn.end();
            iter++ )
      {
        if ( (*iter).dist < (*best).dist )
        {
          best = iter;
        }
      } // for
    } // if
    else
    { // select entry with minimal radius increase
      best = entriesOut.begin();
```

```
        double minIncrease =
            (*best).dist - entriesOut.front().entry->rad();

        list<SearchBestPathEntry>::iterator iter;
        for ( iter = entriesIn.begin();
              iter != entriesIn.end();
              iter++ )
        {
          double increase = (*iter).dist - (*iter).entry->rad();
          if ( increase < minIncrease )
          {
            minIncrease = increase;
            best = iter;
          }
        }

        // update increased covering radius
        (*best).entry->setRad( (*best).dist );
        nodePtr->modified( !SIZE_CHANGED );
      }

      // update path/indizes vector
      path.push_back( (*best).entry->chield() );
      indizes.push_back( (*best).index );

      //load chield node
      deepth++;
      nodePtr->deleteIfAllowed();
      nodePtr = nodeMngr->getNode( (*best).entry->chield() );
  }

  // nodePtr points to a leaf node

  // compute distance to parent node, if exist
  if ( deepth > 0 )
  {
    double dist;
    Node* parent = nodeMngr->getNode( path[deepth-1] );
    Entry* parentEntry = (*parent->getEntries())[indizes[deepth-1]];
    (*metric)( entry->data(), parentEntry->data(), dist );
    entry->setDist( dist );
    parent->deleteIfAllowed();
  }

  // insert entry into leaf, split if neccesary
  if ( !nodePtr->insert( entry ) )
  {
    split( entry );
  }

  nodePtr->deleteIfAllowed();
  header.entryCount++;
}
```

Method *rangeSearch* :

```
void MT::MTree::rangeSearch( Attribute* attr,
                             const double& searchRad,
                             list<TupleId>* results )
{
  #ifdef __MT_DEBUG
  assert( initialized );
  #endif

  results->clear();
  DistData* data = getDistData( attr );

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

  size_t count = 0;
  while( !remainingNodess.empty() )
  {
    RemainingNodesEntry parent = remainingNodess.top();
    nodePtr = nodeMngr->getNode( remainingNodess.top().nodeId );
    unsigned char deepth = remainingNodess.top().deepth;
    double distQueryParent = remainingNodess.top().dist;
    remainingNodess.pop();

    if ( deepth < (header.height - 1) )
    { // routing node
      vector<Entry*>* entries = nodePtr->getEntries();
      for ( size_t i=0; i<entries->size(); i++ )
      {
        Entry* curEntry = (*entries)[i];
        double dist = curEntry->dist();
        double radSum = searchRad + curEntry->rad();
        if ( abs( distQueryParent - dist ) <= radSum )
        {
          double newDistQueryParent;
          (*metric)( data, curEntry->data(), newDistQueryParent );
          if ( newDistQueryParent <= radSum )
          {
            remainingNodess.push( RemainingNodesEntry(
                curEntry->chield(), deepth+1, newDistQueryParent ) );
          }
        }
      }
    }
    else
    { // leaf
      vector<Entry*>* entries = nodePtr->getEntries();
      for ( size_t i=0; i<entries->size(); i++ )
      {
        Entry* curEntry = (*entries)[i];
        double dist = curEntry->dist();
        if ( abs( distQueryParent - dist ) <= searchRad )
```

```
        {
          count++;
          double distQueryCurrent;
          (*metric)( data, curEntry->data(), distQueryCurrent );
          if ( distQueryCurrent <= searchRad )
          {
            results->push_back( curEntry->tid() );
          }
        } // if
      } // for
    } // else
    nodePtr->deleteIfAllowed();
  } // while

  data->deleteIfAllowed();

#ifdef __MT_PRINT_SEARCH_INFO
  cmsg.info() << "Tried " << count << " out of " << header.entryCount
              << " elements..." << endl << endl;
  cmsg.send();
#endif
}
```

Method *rangeSearch* :

```
void MT::MTree::nnSearch( Attribute* attr, int nncount,
                          list<TupleId>* results )
{
// not yet implemented
}
```

2.3 Class *NodeMngr*

December 2007, Mirko Dibbert

2.3.1 Class description

TODO enter class description

2.3.2 Definition part (file: MTNodeMngr.h)

```
#ifndef __MTREE_NODE_MNGR_H
#define __MTREE_NODE_MNGR_H

#include "MTNode.h"

namespace MT
{

class NodeMngr
{
  struct TaggedNode
  {
    TaggedNode()
    : node ( 0 ), tag ( 0 ) {}

    TaggedNode( Node* node_ )
    : node ( node_ ), tag ( NodeMngr::m_tagCntr++ ) {}
```

```
    Node* node;
    unsigned tag;
  };

  map< SmiRecordId, TaggedNode > m_nodes;
  SmiRecordFile* m_file;
  unsigned m_maxNodeEntries;
  unsigned m_hits, m_misses;
  static unsigned m_tagCntr;

  void insert( MT::Node* node );
```

Inserts the node into the cache. If neccesary, the oldest cached node will be replaced.

```
public:
  NodeMngr( SmiRecordFile* file, unsigned maxNodeEntries )
  : m_file ( file ), m_maxNodeEntries ( maxNodeEntries ),
    m_hits( 0 ), m_misses( 0 )
  {}
```

Constructor.

```
  ~NodeMngr()
  {
    map< SmiRecordId, TaggedNode >::iterator iter;

    for(iter = m_nodes.begin(); iter != m_nodes.end(); iter++)
      delete iter->second.node;
    m_nodes.clear();

    #ifdef __MT_PRINT_NODE_CACHE_INFO
    cout << "\nnode-cache hits: " << m_hits
         << ", node-cache misses: " << m_misses << endl;
    #endif
  }
```

Destructor.

```
  MT::Node* getNode( SmiRecordId nodeId );
```

Returns the specified node and increases its ref-count.

```
  MT::Node* createNode();
```

Returns a new node and stores it into cache.

```
}; // class NodeMngr

} // namespace MTree

#endif
```

2.4 Class *MTreeNode*

November 2007, Mirko Dibbert

2.4.1 Class description

TODO enter class description

2.4.2 Definition part (file: MTNode.h)

```
#ifndef __MTREE_NODE_H
#define __MTREE_NODE_H

#include "MTEntry.h"
namespace MT
{

class Node
{

public:
  SmiRecordFile* m_file;  // reference to the m-tree file
  bool m_modified;      // true, if the node has been modified
  unsigned m_maxEntries;  // maximum count of entries per node
  unsigned m_curNodeSize; // current size of the node
  SmiRecordId m_nodeId; // record-id of the node in the m-tree file
  list<SmiRecordId> _extensions; // id's of the extension pages
  vector<Entry*>* m_entries;      // entries stored in this node.
```

```
unsigned char m_refs;

inline Node( SmiRecordFile* file, size_t maxEntries )
: m_file ( file ), m_modified( true ), m_maxEntries ( maxEntries ),
  m_curNodeSize( emptySize() ), m_nodeId ( 0 ), _extensions (),
  m_entries( new vector<Entry*>() ), m_refs( 1 )
{
  #ifdef __MT_DEBUG
  MT::Node::_created++;
  MT::Node::printDebugInfo( true );
  #endif

  m_entries->reserve(
      (NODE_PAGESIZE-emptySize()) / Entry::minSize() + 1 );
}
```

Constructor, creates a new node.

```
inline Node( SmiRecordFile* file, size_t maxEntries,
             SmiRecordId nodeId )
  : m_file ( file ), m_modified( false ),
    m_maxEntries ( maxEntries ), _extensions (),
    m_entries( new vector<Entry*>() ), m_refs( 1 )
{
  #ifdef __MT_DEBUG
  MT::Node::_created++;
  MT::Node::printDebugInfo( true );
  #endif

  read( nodeId );
}
```

Constructor, reads the node from the record `nodeId`.

```
~Node();
```

Destructor.

```
SmiRecordId getNodeId()
{
  if ( !m_nodeId )
  {
    SmiRecord record;
    m_file->AppendRecord( m_nodeId, record );
  }
  return m_nodeId;
}

void removeNode();
```

This method deletes all records of the node from the m-tree file.

```
void remove( vector<Entry*>::iterator iter );
```

Removes entry at position iter from the node.

```
void remove( size_t pos );
```

Removes entry at position pos from the node.

```
inline Node* copy()
{
  if( m_refs == numeric_limits<unsigned char>::max() )
  {
    return new Node( *this );
  }

  m_refs++;
  return this;
}

inline void deleteIfAllowed()
{
  --m_refs;
  if ( !m_refs )
    delete this;
}

void update( size_t pos, Entry* newValue );

inline vector<Entry*>* getEntries()
{
  return m_entries;
}
```

Returns the entry vector (used during split)

```
inline void swapEntries ( vector<Entry*>* entries )
{
  m_entries->swap( *entries );
  m_entries->reserve(
      (NODE_PAGESIZE-emptySize()) / Entry::minSize() + 1);
}
```

Sets new entry vector and returns a pointer to the old vector.

```
inline size_t getEntryCount()
{
  return m_entries->size();
}
```

Returns the count of the currently stored entries.

```
void modified( bool sizeChanged = false );
```

Sets the modified flag to true. If `changedSize == true`, the node size will be recalculated and if nessecary, new extension pages will be added.

```
bool insert( Entry *entry );
```

Tries to inserts an entry into the node and returns true, if succeed. If the method returns false, the node must be splitted.

If entries-¿size() ¡ 2, the node will allways insert the entry into the node. If neccesary, extension pages will be appended to the m-tree file, to get enough space to store the entry.

```
void read( SmiRecordId nodeId );
```

Reads the node from page `nodeId` in the m-tree file.

```
void write();
```

Writes the node to page `nodeId` in the m-tree file.

```
inline static size_t emptySize()
{
  return sizeof( size_t ) + // m_entries->size()
         sizeof( size_t ) + // _extensions.size()
         sizeof( bool );    // _leaf
}
```

Returns the size of an empty node (used to initialize `curNodeSize`)

The following methods are implemented for debugging purposes:

```
#ifdef __MT_DEBUG
private:
  static unsigned _created, _deleted;

public:
  static void printDebugInfo( bool detailed = false )
  {
    #ifdef __MT_PRINT_NODE_INFO
    cmsg.info() << "DEBUG_INFO <MT::NODE> : ";
    if ( detailed )
    {
      cmsg.info() << "objects created : "  << MT::Node::_created
                  << " - objects deleted: " << MT::Node::_deleted
                  << " - ";
    }
    cmsg.info() << "open objects : "  << MT::Node::objectsOpen()
                << endl;
    cmsg.send();
    #endif
  }

  static inline size_t objectsOpen()
  { return ( _created - _deleted ); }
#endif

}; // class Node
```

```
} // namespace MTree

#endif
```

2.4.3 Implementation part (file: MTNode.cpp)

```
#include "MTNode.h"

#ifdef __MT_DEBUG
size_t MT::Node::_created = 0;
size_t MT::Node::_deleted = 0;
#endif
```

Destructor :

```
MT::Node::~Node()
{
#ifdef __MT_DEBUG
  MT::Node::_deleted++;
  MT::Node::printDebugInfo( true );
#endif

  if ( m_modified )
    write();

  for (size_t i=0; i<m_entries->size(); i++)
    delete (*m_entries)[i];

  delete m_entries;
}
```

Method *removeNode* :

```
void MT::Node::removeNode()
{
  list<SmiRecordId>::iterator iter;
  for ( iter = _extensions.begin();
        iter != _extensions.end(); iter++ )
  {
    m_file->DeleteRecord( *iter );
  }

  m_file->DeleteRecord( m_nodeId );
  m_nodeId = 0;
  m_modified = false;
}
```

Method *remove* :

```
void MT::Node::remove( vector<Entry*>::iterator iter )
{
  m_curNodeSize -= ( *iter )->size();
  delete *iter;
  *iter = m_entries->back();
  m_entries->pop_back();
  m_modified = true;
}
```

```
void MT::Node::remove( size_t pos )
{
#ifdef __MT_DEBUG
  assert ( pos < m_entries->size() );
#endif
  m_curNodeSize -= (*m_entries)[pos]->size();
  delete (*m_entries)[ pos ];
  (*m_entries)[pos] = m_entries->back();
  m_entries->pop_back();
  m_modified = true;
}
```

Method *modified* :

```
void MT::Node::modified( bool sizeChanged )
{
  m_modified = true;

  if ( !sizeChanged )
    return;

  // recalculate curSize
  m_curNodeSize = emptySize();
  vector<Entry*>::iterator iter;
  for (iter = m_entries->begin(); iter != m_entries->end(); iter++)
  {
    m_curNodeSize += (*iter)->size();
  }

  // append extension pages, if nessecary
  while (m_curNodeSize > ((_extensions.size()+1) * NODE_PAGESIZE))
  {
    SmiRecordId rec_no;
    SmiRecord rec;
    m_file->AppendRecord( rec_no, rec );
    _extensions.push_back( rec_no );
    m_curNodeSize += sizeof( SmiRecordId );
  }
}
```

Method *insert* :

```
bool
MT::Node::insert( Entry* entry )
{
  if ( m_entries->size() >= m_maxEntries )
    return false;

  size_t newSize = m_curNodeSize + entry->size();
  if ( newSize > ((_extensions.size()+1) * NODE_PAGESIZE) )
  {
    if ( m_entries->size() < 2 )
```

```
    {
      /* Append extension nodeId(s) until the the node is huge
         enough to store the entry. */
#ifdef __MT_DEBUG
      assert ( NODE_PAGESIZE > sizeof( SmiRecordId ) );
#endif
      while (newSize > ((_extensions.size()+1) * NODE_PAGESIZE))
      {
        SmiRecordId rec_no;
        SmiRecord rec;
        m_file->AppendRecord( rec_no, rec );
        _extensions.push_back( rec_no );
        newSize += sizeof( SmiRecordId );
      }

      // insert entry, update curSize
      m_entries->push_back( entry );
      m_curNodeSize = newSize;
      m_modified = true;
      return true;
    }
    else
    {
      return false;
    }
  }

  // insert entry, update curSize
  m_entries->push_back( entry );
  m_curNodeSize = newSize;
  m_modified = true;
  return true;
}
```

Method *update* :

```
void MT::Node::update( size_t pos, Entry* newValue )
{
#ifdef __MT_DEBUG
  assert ( pos < m_entries->size() );
#endif
  m_curNodeSize -= (*m_entries)[ pos ]->size();
  m_curNodeSize += newValue->size();
  delete (*m_entries)[ pos ];
  (*m_entries)[ pos ] = newValue;
  m_modified = true;
}
```

Method *Write* :

```
void MT::Node::write()
{
  if( !m_modified )
```

```
  return;

// open record, if needed append a new page
SmiRecord record;
if ( m_nodeId )
  m_file->SelectRecord( m_nodeId, record, SmiFile::Update);
else
  m_file->AppendRecord( m_nodeId, record);

// remove unneccesary extension nodeIds
while ( m_curNodeSize < ( _extensions.size() * NODE_PAGESIZE ) )
{
  m_file->DeleteRecord( _extensions.back() );
  _extensions.pop_back();
}

// create write buffer
int offset = 0;
int bufferSize =
    ( record.Size() * ( _extensions.size()+1 ) ) +
    ( sizeof( SmiRecordId ) * _extensions.size() );
char buffer[ bufferSize ];

// write number of extension nodeIds
unsigned count = _extensions.size();
memcpy( buffer+offset, &count, sizeof( unsigned ) );
offset += sizeof( unsigned );

// write extension pointer list
list<SmiRecordId>::iterator extIter;
for (extIter = _extensions.begin();
    extIter != _extensions.end(); extIter++ )
{
  memcpy( buffer+offset, &(*extIter), sizeof( SmiRecordId ) );
  offset += sizeof( SmiRecordId );
}

// write number of stored  entries
count = m_entries->size();
memcpy( buffer+offset, &count, sizeof( size_t ) );
offset += sizeof( size_t );

// write the entry array
vector<Entry*>::iterator entryIter;
for ( entryIter = m_entries->begin();
      entryIter != m_entries->end(); entryIter++)
{
  (*entryIter)->write( buffer, offset );
}

record.Write( buffer, record.Size(), 0 );

// write extensions, if exist
```

```
  offset = record.Size();
  for ( extIter = _extensions.begin();
        extIter != _extensions.end(); extIter++ )
  {
    m_file->SelectRecord( *extIter, record, SmiFile::Update );
    record.Write(buffer+offset, record.Size(), 0);
    offset += record.Size();
  }

  // update modified flag
  m_modified = false;
}
```

Method *Read* :

```
void
MT::Node::read( SmiRecordId nodeId  )
{
  if ( m_modified )
    write();

  m_curNodeSize = emptySize();

  SmiRecord record;
  m_nodeId = nodeId;

  // read node (header nodeId)
  char extensionsCountBuf[sizeof( size_t )];
  m_file->SelectRecord( m_nodeId, record, SmiFile::ReadOnly );
  record.Read( extensionsCountBuf, sizeof( size_t ), 0 );

  // read number of extension nodeIds
  size_t extensionsCount;
  memcpy( &extensionsCount, extensionsCountBuf, sizeof( size_t ) );

  int offset = sizeof( size_t );
  int bufferSize =
      ( record.Size() * ( extensionsCount+1 ) ) +
      ( sizeof( SmiRecordId ) * extensionsCount );
  char buffer[ bufferSize ];

  // read node (header nodeId)
  record.Read( buffer, record.Size(), 0 );

  // read extensions, if exist
  int nodeIdoffset = record.Size();
  _extensions.clear();
  for ( size_t i = 0; i < extensionsCount; i++ )
  {
    SmiRecordId rec_no;
    memcpy( &rec_no, buffer+offset, sizeof( SmiRecordId ) );
    _extensions.push_back( rec_no );
    offset += sizeof( SmiRecordId );
```

```
    m_file->SelectRecord( rec_no, record, SmiFile::ReadOnly );
    record.Read( buffer+nodeIdoffset, record.Size(), 0 );
    nodeIdoffset += record.Size();
  }

  // read number of stored entries
  unsigned count;
  memcpy( &count, buffer+offset, sizeof( unsigned ) );
  offset += sizeof( unsigned );

  // delete currently stored entries
  for (size_t i=0; i<m_entries->size(); i++)
    delete (*m_entries)[i];
  m_entries->clear();

  // read the entry array.
  int old_offset = offset;
  m_entries->reserve(
      (NODE_PAGESIZE-emptySize()) / Entry::minSize() );

  unsigned pos = 0;
  while( pos < count )
  {
    pos++;
    m_entries->push_back(new Entry( buffer, offset ) );
  }

  // update size and modified flag
  m_curNodeSize += ( offset - old_offset );
  m_modified = false;

} // read
```

2.5 Class *MT::Entry*

November 2007, Mirko Dibbert

2.5.1 Class description

This class manages the entries, stored in the m-tree nodes and contains the following member functions:

getter	setter	I/O	miscellaneous
dist()	setDist()	write()	minSize()
rad()	setRad()	read()	size()
chield()	setChield()		
tid()	setTid()		
data()			

The write method is used in the write method of the `node` class to write the entry into a buffer. The read method could only be used by calling the appropriate constructor. The size method is used in the `node` class to calculate the current size of the node. The minSize method is only needed to reserve enough space for the entry vector in the `node` class.

Furthermore this class offers two constructors: The first one is used to create a new entry. The second constructor reads the entry from a buffer and is used by the read function of the `node` class.

2.5.2 Definition part (file: MTEntry.h)

```
#ifndef __MT_ENTRY_H
```

```
#define __MT_ENTRY_H

#include "MTreeAlgebra.h"

const unsigned MT_STATIC_ENTRY_SIZE =
    sizeof( TupleId ) +    // tid
    sizeof( double )  +    // dist
    sizeof( double )  +    // rad
    sizeof( SmiRecordId ); // chield


namespace MT
{

class Entry
{
private:
  TupleId m_tid;         // tuple-id of the entry
  double m_dist;         // distance to parent node
  double m_rad;          // covering radius
  SmiRecordId m_chield; // pointer to chield node
  DistData* m_data;      // data string for distance computations
  unsigned m_size;       // actual size of the entry

  void read( const char* const buffer, int& offset );
```

This method reads the entry from `buffer` and increases `offset`.

```
  inline void updateSize()
  {
    m_size = MT_STATIC_ENTRY_SIZE +
             sizeof( size_t ) +      // size of data string
             m_data->size();         // data string
  }
```

This method calculates the actual size of the node.

```
public:
  inline Entry( const TupleId tid, DistData* data )
  : m_tid( tid ), m_dist( 0 ), m_rad ( 0 ), m_chield( 0 ),
    m_data( data )
  {
    #ifdef __MT_DEBUG
    MT::Entry::m_created++;
    MT::Entry::printDebugInfo( true );
    assert ( data );
    #endif

    updateSize();
  }
```

Constructor, creates a new entry object with given tuple id and DistData object.

```
inline Entry( const char* const buffer, int& offset )
{
  #ifdef __MT_DEBUG
  MT::Entry::m_created++;
  MT::Entry::printDebugInfo( true );
  #endif

  read( buffer, offset );
  updateSize();
}
```

Constructor, reads a previously stored entry from `buffer`.

```
inline Entry( const Entry& e )
: m_tid( e.m_tid ), m_dist( e.m_dist ), m_rad ( e.m_rad ),
  m_chield ( e.m_chield ), m_data( e.m_data->copy() ),
  m_size( e.m_size )
{
  #ifdef __MT_DEBUG
  MT::Entry::m_created++;
  MT::Entry::printDebugInfo( true );
  #endif
}
```

Copy constructor.

```
inline ~Entry()
{
  m_data->deleteIfAllowed();

  #ifdef __MT_DEBUG
  MT::Entry::m_deleted++;
  MT::Entry::printDebugInfo( true );
  #endif
}
```

Destructor.

```
inline const DistData* data() const
{ return m_data; }
```

Returns the data object.

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

Returns the tid value.

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

Returns distance to parent node.

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

Returns the covering radius.

```
inline SmiRecordId chield() const
{ return m_chield; }
```

Returns record id of the chield node.

```
inline void setTid( TupleId tid )
{ m_tid = tid; }
```

Sets a new value for the tuple id.

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

Sets distance to parent node.

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

Sets a new covering radius.

```
void setChield( const SmiRecordId chield )
{ m_chield = chield; }
```

Sets a new chield node.

```
Entry& operator=( const Entry& e );
```

Assignment operator.

```
static size_t minSize();
```

This method returns the minimal size of an entry on disc and is used in the constructors of the `Node` class to reserve an adequate amount of memory for the entry vector.

```
size_t size() const;
```

This method returns the actual size of the entry on disc.

```
void write( char* const buffer, int& offset ) const;
```

This method writes the entry to `buffer` and increases `offset`.

The following methods are implemented for debugging purposes:

```
#ifdef __MT_DEBUG
private:
  static unsigned m_created, m_deleted;

public:
  static void printDebugInfo( bool detailed = false )
  {
  #ifdef __MT_PRINT_ENTRY_INFO
    cmsg.info() << "DEBUG_INFO <MT::ENTRY> : ";
    if ( detailed )
    {
      cmsg.info() << "objects created : "  << MT::Entry::m_created
                  << " - objects deleted: " << MT::Entry::m_deleted
                  << " - ";
    }
    cmsg.info() << "open objects : "  << MT::Entry::objectsOpen()
                << endl;
    cmsg.send();
  #endif
  }

  static inline size_t objectsOpen()
  { return ( m_created - m_deleted ); }
#endif

}; // class Entry

} // namespace MTree

#endif
```

2.5.3 Implementation part (file: MTEntry.cpp)

```
#include "MTEntry.h"
```

Initialise static members :

```
#ifdef __MT_DEBUG
size_t MT::Entry::m_created = 0;
size_t MT::Entry::m_deleted = 0;
#endif
```

Assignment Operator :

```
MT::Entry&
MT::Entry::operator=( const MT::Entry& e )
{
  m_tid = e.m_tid;
  m_dist = e.m_dist;
  m_rad = e.m_rad;
  m_chield = e.m_chield;

  // copy e.m_data
  DistData* tmp = new DistData( *e.m_data );
  if ( m_data )
    delete m_data;
  m_data = tmp;

  return* this;
}
```

Method *minSize* :

```
size_t
MT::Entry::minSize()
{
  return sizeof( TupleId ) +    // tid
         sizeof( double )  +    // dist
         sizeof( double )  +    // rad
         sizeof( SmiRecordId ); // chield
}
```

Method *size* :

```
size_t
MT::Entry::size() const
{
  return m_size;
}
```

Method *write* :

```
void
MT::Entry::write( char* const buffer, int& offset ) const
{
  // write tid, dist, rad and chield
  memcpy( buffer+offset, this, MT_STATIC_ENTRY_SIZE );
  offset += MT_STATIC_ENTRY_SIZE;

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

Method *read* :

```
void
MT::Entry::read( const char* const buffer, int& offset )
{
  // read tid, dist, rad and chield
  memcpy( this, buffer+offset, MT_STATIC_ENTRY_SIZE );
  offset += MT_STATIC_ENTRY_SIZE;

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

2.6 Class *Splitpol*

December 2007, Mirko Dibbert

2.6.1 Class description

TODO enter class description

2.6.2 Definition part (file: MTSplitpol.h)

```
#ifndef SPLITPOL_H
#define SPLITPOL_H

#include "MTNode.h"

namespace MT
{

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

Enumeration of the implemented promote functions:

- `RANDOM` : This Algorithm promotes two random entries.
- `m_RAD` : TODO
- `mM_RAD` : TODO

- `M_LB_DIST` : TODO

  ```
  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.

  ```
  class Splitpol
  {
    vector<Entry*>* m_entries;  // contains the original entry-vector
    vector<Entry*>* m_entriesL; // return vector for left node
    vector<Entry*>* m_entriesR; // return vector for right node
    Entry* m_promL;             // promoted Entry for left node
    Entry* m_promR;             // promoted Entry for right node
    size_t m_promLId;           // index of the left promoted entry
    size_t m_promRId;           // index of the right promoted entry
    double m_radL, m_radR;      // covering radii of the prom-entries
    bool m_isLeaf;              // true, if the splitted node is a leaf

    vector< vector<double> > m_distances;
    bool m_distancesDefined;

    TMetric m_metric;
  ```

Contains the selected metric.

```
    void ( Splitpol::*promFun )();
```

Contains the selected promote function.

```
    void ( Splitpol::*partFun )();
```

Contains the selected partiton function.

```
  struct BalancedPromEntry
  {
    Entry* entry;
    double distToL, distToR;

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

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

```
public:
  Splitpol( PROMOTE promId, PARTITION partId, TMetric metric );
```

Constructor.

```
    inline void apply( vector<Entry*>* entries,
                       vector<Entry*>* entriesL,
                       vector<Entry*>* entriesR,
                       bool isLeaf )
    {
      m_entries = entries;
      m_entriesL = entriesL;
      m_entriesR = entriesR;
      m_isLeaf = isLeaf;

      m_distancesDefined = false;

      (this->*promFun)();
      (this->*partFun)();
      m_promL = new Entry( *((*entries)[ m_promLId ]) );
      m_promR = new Entry( *((*entries)[ m_promRId ]) );
      m_promL->setRad( m_radL );
      m_promR->setRad( m_radR );
    }
```

This function applies the split policy, which had been selected in the constructor, to the entry lists. As result there are two lists created in the supp object, which could be obtained by the respective methods.

```
    inline Entry* getPromL()
    {
      return m_promL;
    }

    inline Entry* getPromR()
    {
      return m_promR;
    }

  private:
```

Promote functions:

The following methods promote two objects in the entries list and return them in `promL` and `promR`. The promoted entries will be deleted from entries list and returned in `promL` and `promR`.

```
    void Rand_Prom();
```

This method promtes two randomly selected elements.

```
    void MRad_Prom();
```

TODO : enter method description

```
void MMRad_Prom();
```

TODO : enter method description

```
void MLB_Prom();
```

TODO : enter method description

Partition functions:

The following methods divide the entries of the first list into two lists, which would be returned in `entries1` and `entries2`. The covering radii of the new lists will be returned in rad1 and rad2, respectively.

```
void Hyperplane_Part();
```

TODO : enter method description

```
void Balanced_Part();
```

TODO : enter method description

```
}; // class Splitpol

} // namespace MTree

#endif
```

2.6.3 Implementation part (file: MTSplitpol.cpp)

```
#include "MTSplitpol.h"
```

Constructor :

```
MT::Splitpol::Splitpol( PROMOTE promId, PARTITION partId,
                        TMetric metric )
{
  m_metric = metric;
  srand( time(0) ); // needed for Rand_Prom
  unsigned maxEntries =
      ((NODE_PAGESIZE - Node::emptySize()) / Entry::minSize()) + 1;
  m_distances.reserve( maxEntries );
  for (unsigned i=0; i<maxEntries; i++)
  {
    m_distances.push_back( vector<double>() );
    m_distances.back().reserve( maxEntries - i );
  }
  // init promote function

  switch ( promId )
  {

    case RANDOM:
      promFun = &Splitpol::Rand_Prom;
      break;

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

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

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

  // init partition function
  switch ( partId )
  {

    case GENERALIZED_HYPERPLANE:
      partFun = &Splitpol::Hyperplane_Part;
      break;

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

```
}
```

Method *RandProm* :

```
void MT::Splitpol::Rand_Prom()
{
//   assert ( m_entries->size() >= 2 );
  unsigned pos1 = rand() % m_entries->size();
  unsigned pos2 = rand() % m_entries->size();
  if (pos1 == pos2)
  {
    if ( pos1 == 0 )
      pos1++;
    else
      pos1--;
  }

  if ( pos1 > pos2 )
    std::swap( pos1, pos2 );

  m_promLId = pos1;
  m_promRId = pos2;
}
```

Method *MRad_Prom* :

```
void MT::Splitpol::MRad_Prom()
{
  bool first = true;
  unsigned bestProm1 = 0;
  unsigned bestProm2 = 1;
  double minRadSum;

  vector<Entry*>::iterator prom1Iter;
  vector<Entry*>::iterator prom2Iter;
  vector<Entry*>::iterator last = (m_entries->end())--;

  unsigned i = 0;
  prom1Iter = m_entries->begin();
  while ( prom1Iter != last )
  {
    unsigned j = i + 1;
    prom2Iter = prom1Iter;
    prom2Iter++;
    m_distances[ i ].clear();
    while ( prom2Iter != m_entries->end() )
    {
      double dist;
      (*m_metric)
          ((*m_entries)[ i ]->data(),
           (*m_entries)[ j ]->data(), dist);
      m_distances[ i ].push_back( dist );
```

```
      j++;
      prom2Iter++;
    }

    i++;
    prom1Iter++;
  }
  m_distancesDefined = true;

  i = 0;
  prom1Iter = m_entries->begin();
  while ( prom1Iter != last )
  {
    unsigned j = i + 1;
    prom2Iter = prom1Iter;
    prom2Iter++;
    while ( prom2Iter != m_entries->end() )
    {
      m_promLId = i;
      m_promRId = j;

      (this->*partFun)();
      if ( first )
      {
        minRadSum = ( m_radL + m_radR );
        first = false;
      }
      else
      {
        if ( ( m_radL + m_radR ) < minRadSum )
        {
          minRadSum = ( m_radL + m_radR );
          bestProm1 = i;
          bestProm2 = j;
        }
      }

      j++;
      prom2Iter++;
    }

    i++;
    prom1Iter++;
  }
  m_promLId = bestProm1;
  m_promRId = bestProm2;
}
```

Method *MMRadProm* :

```
void MT::Splitpol::MMRad_Prom()
{
  bool first = true;
```

```
unsigned bestProm1 = 0;
unsigned bestProm2 = 1;
double minMaxRad;

vector<Entry*>::iterator prom1Iter;
vector<Entry*>::iterator prom2Iter;
vector<Entry*>::iterator last = (m_entries->end())--;

unsigned i = 0;
prom1Iter = m_entries->begin();
while ( prom1Iter != last )
{
  unsigned j = i + 1;
  prom2Iter = prom1Iter;
  prom2Iter++;
  m_distances[ i ].clear();
  while ( prom2Iter != m_entries->end() )
  {
    double dist;
    (*m_metric) (
        (*m_entries)[ i ]->data(),
        (*m_entries)[ j ]->data(), dist );
    m_distances[ i ].push_back( dist );

    j++;
    prom2Iter++;
  }

  i++;
  prom1Iter++;
}
m_distancesDefined = true;

i = 0;
prom1Iter = m_entries->begin();
while ( prom1Iter != last )
{
  unsigned j = i + 1;
  prom2Iter = prom1Iter;
  prom2Iter++;
  while ( prom2Iter != m_entries->end() )
  {
    m_promLId = i;
    m_promRId = j;

    (this->*partFun)();
    if ( first )
    {
      minMaxRad = max( m_radL, m_radR );
      first = false;
    }
    else
    {
```

```
          if ( max( m_radL, m_radR ) < minMaxRad )
          {
            minMaxRad = max( m_radL, m_radR );
            bestProm1 = i;
            bestProm2 = j;
          }
        }

        j++;
        prom2Iter++;
      }

      i++;
      prom1Iter++;
    }
    m_promLId = bestProm1;
    m_promRId = bestProm2;
  }
```

Method *MLBProm* :

```
void MT::Splitpol::MLB_Prom()
{
  // TODO : not yet implemented
  assert( false );
}
```

Method *HyperplanePart* :

```
void MT::Splitpol::Hyperplane_Part()
{
  m_entriesL->clear();
  m_entriesR->clear();

  m_entriesL->push_back( (*m_entries)[ m_promLId ] );
  m_entriesR->push_back( (*m_entries)[ m_promRId ] );

  m_radL = 0;
  m_radR = 0;

  double distL, distR;
  (*m_entries)[ m_promLId ]->setDist( 0 );
  (*m_entries)[ m_promRId ]->setDist( 0 );

  if ( !m_isLeaf )
    m_radL = max( m_radL, (*m_entries)[ m_promLId ]->rad() );

  if ( !m_isLeaf )
    m_radR = max( m_radR, (*m_entries)[ m_promRId ]->rad() );

  for ( size_t i=0; i<m_entries->size(); i++ )
  {
    if ( (i != m_promLId) && (i != m_promRId) )
```

```
        {
          // TODO wenn vorhanden, zuvor berechnete Distanzen nutzen!
          (*m_metric)( (*m_entries)[ i ]->data(),
                              (*m_entries)[ m_promLId ]->data(), distL );

          (*m_metric)( (*m_entries)[ i ]->data(),
                              (*m_entries)[ m_promRId ]->data(), distR );

          if ( distL < distR )
          {
            if ( m_isLeaf )
              m_radL = max( m_radL, distL );
            else
              m_radL = max( m_radL, distL + (*m_entries)[ i ]->rad() );

            m_entriesL->push_back( (*m_entries)[i] );
            m_entriesL->back()->setDist( distL );
          }
          else
          {
            if ( m_isLeaf )
              m_radR = max( m_radR, distR );
            else
              m_radR = max( m_radR, distR + (*m_entries)[ i ]->rad() );

            m_entriesR->push_back( (*m_entries)[i] );
            m_entriesR->back()->setDist( distR );
          }
        }
      }
    }
```

Method *BalancedPart* :

```
void MT::Splitpol::Balanced_Part()
{
  m_entriesL->clear();
  m_entriesR->clear();

  m_entriesL->push_back( (*m_entries)[ m_promLId ] );
  m_entriesR->push_back( (*m_entries)[ m_promRId ] );

  m_radL = 0;
  m_radR = 0;

  (*m_entries)[ m_promLId ]->setDist( 0 );
  (*m_entries)[ m_promRId ]->setDist( 0 );

  if ( !m_isLeaf )
    m_radL = max( m_radL, (*m_entries)[ m_promLId ]->rad() );

  if ( !m_isLeaf )
    m_radR = max( m_radR, (*m_entries)[ m_promRId ]->rad() );
```

```
  list<BalancedPromEntry> entries;
  for ( size_t i=0; i<m_entries->size(); i++ )
  {
    if ( (i != m_promLId) && (i != m_promRId) )
    {
      double distL;
      double distR;

      if ( m_distancesDefined )
      {
        if ( i < m_promLId )
          distL = m_distances[ i ][ m_promLId-(i+1) ];
        else
          distL = m_distances[ m_promLId ][ i-(m_promLId+1) ];

        if ( i < m_promRId )
          distR = m_distances[ i ][ m_promRId-(i+1) ];
        else
          distR = m_distances[ m_promRId ][ i-(m_promRId+1) ];

      }
      else
      {
        (*m_metric)( ((*m_entries)[ i ])->data(),
                     ((*m_entries)[ m_promLId ])->data(), distL );
        (*m_metric)( ((*m_entries)[ i ])->data(),
                     ((*m_entries)[ m_promRId ])->data(), distR );
      }
      entries.push_back(
          BalancedPromEntry(
              ((*m_entries)[ i ]), distL, distR));
//          (*m_metric)( (*m_entries)[ i ]->data(),
//                       (*m_entries)[ m_promLId ], distL );
//          (*m_metric)( (*m_entries)[ i ]->data(),
//                       (*m_entries)[ m_promRId ], distR );
//          assert ( distL == entries.back().distToL );
//          assert ( distR == entries.back().distToR );
    }
  }

  bool assignLeft = true;
  while ( !entries.empty() )
  {
    if ( assignLeft )
    {
      list<BalancedPromEntry>::iterator nearestPos = entries.begin();
      list<BalancedPromEntry>::iterator iter = entries.begin();
      while ( iter  != entries.end() )
      {
        if ( (*iter).distToL < (*nearestPos).distToL )
        {
          nearestPos = iter;
```

```
      }
      iter++;
    }

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

    m_entriesL->push_back( (*nearestPos).entry );
    m_entriesL->back()->setDist( distL );
    entries.erase ( nearestPos );
  }
  else
  {
    list<BalancedPromEntry>::iterator nearestPos = entries.begin();
    list<BalancedPromEntry>::iterator iter = entries.begin();
    while ( iter  != entries.end() )
    {
      if ( (*iter).distToL < (*nearestPos).distToR )
      {
        nearestPos = iter;
      }
      iter++;
    }
    double distR = (*nearestPos).distToR;
    if ( m_isLeaf )
      m_radR = max( m_radR, distR );
    else
      m_radR = max( m_radR, distR + (*nearestPos).entry->rad() );

    m_entriesR->push_back( (*nearestPos).entry );
    m_entriesR->back()->setDist( distR );

    entries.erase ( nearestPos );
  }
    assignLeft = !assignLeft;
 }
}
```

3 Managing Metrics

December 2007, Mirko Dibbert

3.1 Overview

Every type constructor, which needs a metric (e.g. to be indexed by m-trees), has to implement at least one method of the type `TMetric` for the respective type constructor in the class `MetricRegistry` (see below). The metrics should except DistData objects, which are created with the `getDistData` method of the respective attribute class, which must inherrit from `MetricalAttribute` (extends `IndexableStandardAttribute` to provide this method.

3.2 Class *MetricRegistry*

3.2.1 Class description

TODO enter class description

3.2.2 Definition part (file: MetricRegistry.h)

```
#ifndef __METRIC_REGISTRY_H
#define __METRIC_REGISTRY_H

#define DEBUG_METRIC_REGISTRY
```

```
#include <string>
#include <map>
#include "SecondoInterface.h"

const string MF_DEFAULT = "default";
```

The name for default metrics. Each type constructor which need a metric, should define one of the provided metrics as default metric, which is used if no metric is specified.

```
typedef void ( *TMetric )( const void* data1, const void* data2,
               double& result );
```

Type definition for metrics.

```
class MetricRegistry
{
  struct MetricData
  {
    string tcName;
    TMetric metric;
    string descr;

    MetricData()
    {}

    inline MetricData( const string& tcName_,
               const TMetric metric_,
               const string& descr_ )
    : tcName ( tcName_ ), metric ( metric_ ), descr ( descr_ )
    {}
  }; // MetricData

  static map< string, MetricData > metric_map;
  static bool initialized;

  static void registerMetric( const string& metricName,
               const MetricData& data );
```

This method is used to register a new metric.

```
  static void initialize();
```

This method registeres all defined distance functions.

```
public:
  static TMetric getMetric( const string& tcName,
                            const string& metricName );
```

This method returns the associated distance function (0, if no distance function was found).

```
  static ListExpr listMetrics();
```

This method returns all registered metrics in a list, wich has the following fomrat:

((tcName metricName metricType metricDescr)...(...))

This list is used in the `DisplayTTY` class to print the registered metrics in a formated manner, which is used by the `list metrics` command.

```
private:
```

Below, all avaliable metrics will be defined:

```
  static void EuclideanInt(
      const void* data1, const void* data2, double& result );
```

Euclidean distance function for the `int` type constructor.

```
  static void EuclideanReal(
      const void* data1, const void* data2, double& result );
```

Euclidean distance function for the `real` type constructor.

```
  static void EditDistance(
      const void* data1, const void* data2, double& result );
```

Edit distance function for the `string` type constructor.

```
  static void HistogramMetric(
      const void* data1, const void* data2, double& result );
```

Metric for the `histogram` type constructor.

```
  static void PictureMetric(
      const void* data1, const void* data2, double& result );
```

Metric for the `picture` type constructor.

```
};

#endif
```

3.2.3 Implementation Part (file: MetricRegistry.cpp)

```
using namespace std;

#include <math.h>
#include <sstream>
#include "NList.h"
#include "StandardTypes.h"
#include "MetricalAttribute.h"
#include "MetricRegistry.h"
#include "StandardTypes.h"
#include "PictureAlgebra.h"

extern SecondoInterface* si;
```

Initialize static members :

```
bool MetricRegistry::initialized = false;
map< string, MetricRegistry::MetricData > MetricRegistry::metric_map;
```

Method *registerMetric* :

The default metric will be stored without a name to ensure that this metric is allways the first one for each type constructor, which `listMetrics` put into the result list. The algebra- and type-id are only used to order the output of `listMetrics` by these id's.

```
void
MetricRegistry::registerMetric( const string& metricName,
                const MetricData& data )
{
  int algebraId, typeId;
  si->GetTypeId( data.tcName, algebraId, typeId );

  ostringstream osId;
  osId << algebraId << "#" << typeId << ".";
  if ( metricName != MF_DEFAULT )
    osId << metricName;

  metric_map[osId.str()] = data;
}
```

Method *getMetric* :

```
TMetric
MetricRegistry::getMetric( const string& tcName,
               const string& metricName )
{
  if (!initialized)
    initialize();

  int algebraId, typeId;
  si->GetTypeId( tcName, algebraId, typeId );

  ostringstream osId;
  osId << algebraId << "#" << typeId << ".";
  if ( metricName != MF_DEFAULT )
    osId << metricName;

  map< string, MetricData >::iterator pos =
      metric_map.find( osId.str() );

  if ( pos != metric_map.end() )
  {
    return pos->second.metric;
  }
  else return 0;
}
```

Method *ListMetrics* :

```
ListExpr
MetricRegistry::listMetrics()
{
  if (!initialized)
    initialize();

  NList list;
  NList elem;
  ostringstream os;

  map< string, MetricData >::iterator pos = metric_map.begin();
  while ( pos != metric_map.end() )
  {
    string key = pos->first;

    // get metricName
    string metricName = key.substr( key.find( '.' ) + 1 );
    if ( metricName == "" )
      metricName = MF_DEFAULT;

    // get tcName
    string tcName = pos->second.tcName;

    // append item list to the output list
    NList e1( tcName );
    NList e2( metricName );
    NList e3 = e3.textAtom(pos->second.descr);
    list.append( NList( e1, e2, e3 ) );
    pos++;
  };

  return list.listExpr();
}
```

Below, the avaliable metrics will be implemented:

Method *EuclideanInt* :

```
void MetricRegistry::EuclideanInt(
    const void* data1, const void* data2, double& result )
{
  int val1 = *static_cast<const int*>
      ( static_cast<const DistData*>( data1 )->value() );

  int val2 = *static_cast<const int*>
      ( static_cast<const DistData*>( data2 )->value() );

  result = abs( val1 - val2 );
}
```

Method *EuclideanReal* :

```
void MetricRegistry::EuclideanReal(
    const void* data1, const void* data2, double& result )
{
  SEC_STD_REAL val1 = *static_cast<const SEC_STD_REAL*>
      ( static_cast<const DistData*>( data1 )->value() );

  SEC_STD_REAL val2 = *static_cast<const SEC_STD_REAL*>
      ( static_cast<const DistData*>( data2 )->value() );

  result = abs( val1 - val2 );
}
```

Method *EditDistance* :

```
void MetricRegistry::EditDistance(
    const void* data1, const void* data2, double& result )
{
  const char* str1 = static_cast<const char*>
      ( static_cast<const DistData*>( data1 )->value() );
  const char* str2 = static_cast<const char*>
      ( static_cast<const DistData*>( data2 )->value() );

  int len1 = static_cast<const DistData*>( data1 )->size();
  int len2 = static_cast<const DistData*>( data2 )->size();

  int d[len1 + 1][len2 + 1];
  int dist;

  // init row 1 with
  for ( int i = 0; i <= len1; i++ )
    d[i][0] = i;

  // init col 1
  for ( int j = 1; j <= len2; j++ )
    d[0][j] = j;

  // compute array getValues
  for ( int i = 1; i <= len1; i++ )
  {
    for ( int j = 1; j <= len2; j++ )
    {
      if ( str1[i - 1] == str2[j - 1] )
        dist = 0;
      else
        dist = 1;

      // d(i,j) = min{ d( i-1 , j   ) + 1,
      //         d( i   , j-1 ) + 1,
      //         d( i-1 , j-1 ) + dist }
      d[i][j] = min( d[i - 1][j] + 1,
                min(( d[i][j - 1] ) + 1,
                 d[i - 1][j - 1] + dist ) );
    }
```

```
    }
    result = ( double ) d[len1][len2];
  }
```

Method *HistogramMetric* :

```
void MetricRegistry::HistogramMetric(
    const void* data1, const void* data2, double& result )
{
  // TODO compute result value
}
```

Method *PictureMetric* :

```
void MetricRegistry::PictureMetric(
    const void* data1, const void* data2, double& result )
{
  const double* values1 = static_cast<const double*>
      ( static_cast<const DistData*>( data1 )->value() );

  const double* values2 = static_cast<const double*>
      ( static_cast<const DistData*>( data2 )->value() );

  result = 0;
  for (int i=0; i<512; i++)
   result += pow(( values1[i] - values2[i] ), 2);
  result = sqrt(result);
  cout << result << endl;
}
```

Method *Initialize* :

Insert a call of `registerMetric` in the following method for every metric, that should be avaliable for the using algebras. The `registerMetric` parameter have the folowing meanings:

1. Name of the metric (must be unique for every type constructor)
2. `MF_Data` object, which contains the neccesary data for the metric

The MF_Data constructor takes the following parameter:

1. Name of the associated type constructor.
2. Reference to the method, which implements the metric.
3. Parameter type (`DF_DATA` or `DF_REFERENCE`)
4. Description of the metric

```
void
MetricRegistry::initialize()
{
  //int type constructor
  registerMetric( MF_DEFAULT,
      MetricData( "int",& EuclideanInt,
      "Euclidean distance metric" ));

  // real type constructor
  registerMetric( MF_DEFAULT,
      MetricData( "real",& EuclideanReal,
      "Euclidean distance metric" ));

  // string type constructor
  registerMetric( MF_DEFAULT,
      MetricData( "string",& EditDistance,
      "Edit distance metric" ));

  // string type constructor
  registerMetric( "EditDist1",
      MetricData( "string", &EditDistance,
      "Edit distance metric (alternative MTreeConfig: "
      "minimum rad prom, balanced part )" ));

  // string type constructor
  registerMetric( "EditDist2",
      MetricData( "string", &EditDistance,
      "Edit distance metric (alternative MTreeConfig: "
      "Random prom, balanced part)" ));

  // histogram type constructor
  registerMetric( MF_DEFAULT,
      MetricData( "histogram",& HistogramMetric,
      "Not yet implemented" ));

  // picture type constructor
  registerMetric( MF_DEFAULT,
      MetricData( "picture",& PictureMetric,
      "Not yet implemented" ));
}
```

3.3 Class *MetricalAttribute*

December 2007, Mirko Dibbert

3.3.1 Class description

This interface class provide a new method, which is needed to obtain a `DistData` object from an attribute object. These objects will be stored within m-trees and are used as parameter objects of the respective metric.

3.3.2 Definition part (file: MetricalAttribute.h)

```
#ifndef __METRICAL_ATTRIBUTE_H
#define __METRICAL_ATTRIBUTE_H

#include "StandardAttribute.h"
#include "DistData.h"

class MetricalAttribute
: public IndexableStandardAttribute
{

public:
  virtual DistData* getDistData( const string& metricName ) = 0;
```

This method should return a new DistData object, which must correspond with the DistData object that the respective metric(defined in the class `MetricRegistry`) excepts.

The `metricName` parameter may be used, if the attribute should return different strings for different metrics (e.g. one metric, which expects value vectors, and another, which expects two filenames and restores the value vectors from these files).

```
}; // MetricalAtrubite
```

```
#endif
```

3.4 Class *DistData*

December 2007, Mirko Dibbert

3.4.1 Class description

This class contains a data array, which contains all neccecary data for distance computations. For each metric, the respective objects will be created with the `getDistData` method of the corresponding attribute class.

3.4.2 Definition part (file: DistData.h)

```
#ifndef __DISTDATA_H
#define __DISTDATA_H

// #define __DEBUG_DISTDATA

#include <iostream>
#include <string>
#include "assert.h"
#include "LogMsg.h"

class DistData
{
  size_t m_size;
  char* m_value;
  unsigned char m_refs;
```

```
public:
  inline DistData( size_t size, const void* value )
  : m_size( size ), m_value( new char[size] ), m_refs( 1 )
  {
    memcpy( m_value, value , m_size );

    #ifdef __DEBUG_DISTDATA
    DistData::m_created++;
    #endif
  }
```

Constructor, creates a new object with length `size` and read it's value from `value`.

```
  inline DistData( const char* buffer, int& offset )
  : m_refs ( 1 )
  {
    memcpy( &m_size, buffer + offset, sizeof(size_t) );
    offset += sizeof(size_t);

    m_value = new char[m_size];
    memcpy( m_value, buffer + offset, m_size );
    offset += m_size;

#ifdef __DEBUG_DISTDATA
    DistData::m_created++;
#endif
  }
```

Read constructor, creates a new object and read it's size and value from buffer, starting at position offset - offset is increased.

```
  inline DistData( const string value )
  : m_size( value.size() ), m_value( new char[m_size] ), m_refs( 1 )
  {
    memcpy( m_value, value.c_str(), m_size );

#ifdef __DEBUG_DISTDATA
    DistData::m_created++;
#endif
  }
```

Constructor, creates a new object from a string.

```
  inline DistData( const DistData& e )
  : m_size ( e.m_size ), m_value( new char[e.m_size] ), m_refs( 1 )
  {
    memcpy( m_value, e.m_value, e.m_size );

#ifdef __DEBUG_DISTDATA
    DistData::m_created++;
#endif
  }
```

Copy constructor.

```
  inline ~DistData()
  {
    delete m_value;

#ifdef __DEBUG_DISTDATA
    assert( !m_refs );
    DistData::m_deleted++;
#endif
  }
```

The Destructor.

```
  inline DistData* copy()
  {
    if( m_refs == numeric_limits<unsigned char>::max() )
      return new DistData( *this );

    m_refs++;
    return this;
  }

  inline void deleteIfAllowed()
  {
    --m_refs;
    if ( !m_refs )
      delete this;
  }

  inline const void* value() const
  { return m_value; }
```

Returns `m_value`.

```
  inline size_t size() const
  { return m_size; }
```

Returns `m_size`.

```
  DistData& operator=( const DistData& e );
```

Assignment Operator.

```
  void write( char* buffer, int& offset ) const;
```

Writes the data string to the buffer at position offset. Offset is increased.

```
#ifdef __DEBUG_DISTDATA
```

The following methods are implemented for debugging purposes:

```
private:
  static size_t m_created, m_deleted;

public:
  static inline size_t created() const
  { return m_created; }

  static inline size_t deleted() const
  { return m_deleted; }

  static inline size_t openObjects() const
  { return ( m_created - m_deleted ); }
#endif

}; // class DistData

#endif
```

3.4.3 Implementation part (file: DistData.cpp)

```
#include "DistData.h"
```

Initialisation of static members :

```
#ifdef __DEBUG_DISTDATA
size_t DistData::m_created = 0;
size_t DistData::m_deleted = 0;
#endif
```

Assignment Operator :

```
DistData&
DistData::operator=( const DistData& e )
{
  m_size = e.m_size;
  char* newValue = new char[ e.m_size ];

  memcpy( newValue, e.m_value, e.m_size );
  delete m_value;
  m_value = newValue;

  return* this;
}
```

Method *write* :

```
void
DistData::write( char* buffer, int& offset ) const
{
  // write m_size
  memcpy( buffer + offset, &m_size, sizeof( size_t ) );
  offset += sizeof( size_t );

  // write m_value
  memcpy( buffer + offset, m_value, m_size );
  offset += m_size;
}
```

3.5 Configuring m-trees

December 2007, Mirko Dibbert

3.5.1 Overview

TODO

3.5.2 Definition part (file: MTreeConfig.h)

```
#include <string>
#include <map>
#include "MTSplitpol.h"

namespace MT
{
```

Struct *MTreeConfig* :
This struct contains some config parameter, which allows it to optimize the mtree datastructure.

```
struct MTreeConfig
{
  unsigned maxNodeEntries;
```

This parameter adjust the maximum count of entries, wich should be stored within a m-tree node when the associated metric is used.

A limiting value could make sense, if the DistData values are very short and the cost of distance computations is (much) higher than the cost of the additional I/O access duo to the growing count of nodes.

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

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

```
MTreeConfig()
: maxNodeEntries ( 200 ),
  promoteFun( RANDOM ),
  partitionFun( BALANCED ) {}
```

Constructor (creates object with default values).

```
MTreeConfig( unsigned maxNodeEntries_,
             PROMOTE promoteFun_,
             PARTITION partitionFun_ )
: maxNodeEntries ( (maxNodeEntries_ < 2) ? 2 : maxNodeEntries_ ),
  promoteFun( promoteFun_ ),
  partitionFun( partitionFun_ ) {}
```

Constructor (creates objects with the given parameters).

```
};
```

Class *MTreeConfigReg* :
TODO insert description

```
class MTreeConfigReg
{
  static map< string, MTreeConfig > mTreeConfig_map;
  static bool initialized;

public:
  static MTreeConfig getMTreeConfig( const string& name );
```

This method returns the MTreeConfig object, that belongs to the specified metric. If no such object is registered, the method returns a new object with default values.

```
  static void initialize();
```

This method registeres all defined distance functions.

```
  static void registerMTreeConfig( const string& name,
                                   const MTreeConfig& config );
```

This method is used to register a `MTreeConfig` object for the associatedmetric.

```
};

} // namespace
```

3.5.3 Implementation part (file: MTreeConfig.cpp)

```
#include "MTreeAlgebra.h"
#include "MTreeConfig.h"
```

Initialise static members :

```
bool MT::MTreeConfigReg::initialized = false;
map< string, MT::MTreeConfig > MT::MTreeConfigReg::mTreeConfig_map;
```

Method *registerMTreeConfig* :

```
void
MT::MTreeConfigReg::registerMTreeConfig( const string& name,
                                         const MTreeConfig& config )
{
  mTreeConfig_map[ name ] = config;
}
```

Method *getMTreeConfig* :

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

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

  if ( pos != mTreeConfig_map.end() )
  {
    #ifdef __MT_PRINT_CONFIG_INFO
    string promFunStr, partFunStr;
    switch ( pos->second.promoteFun )
    {
      case RANDOM:
        promFunStr = "random";
        break;
      case m_RAD:
        promFunStr = "minmal sum of covering radii";
        break;
      case mM_RAD:
        promFunStr = "minimal maximum of covering radii";
        break;
      case M_LB_DIST:
        promFunStr = "maximum lower bound on distance";
        break;
    }
    switch ( pos->second.partitionFun )
    {
      case GENERALIZED_HYPERPLANE:
        partFunStr = "generalized hyperplane";
```

```
        break;
      case BALANCED:
        partFunStr = "balanced";
        break;
    }
    cmsg.info() << endl
                << "Found mtree-config: " << endl
                << "-----------------------------------------" << endl
                << "max entries per node: "
                << pos->second.maxNodeEntries << endl
                << "promote function: " << promFunStr << endl
                << "partition function: " << partFunStr << endl
                << endl;
    cmsg.send();
    #endif
    return pos->second;
  }
  else
  {
    #ifdef MT_PRINT_CONFIG_INFO
    cmsg.info() << "No mtree-config found, using default values."
                << endl;
    cmsg.send();
    #endif
    return MTreeConfig();
  }
}
```

Method *initialize* :

```
void
MT::MTreeConfigReg::initialize()
{
  registerMTreeConfig( "default",  MTreeConfig() );

  registerMTreeConfig( "rand",
      MTreeConfig(
          80,      // maxNodeEntries
          RANDOM,  // promote function (min. covering radius)
          BALANCED // partition function
      ));

  registerMTreeConfig( "mRad",
      MTreeConfig(
          80,      // maxNodeEntries
          m_RAD,   // promote function (min. covering radius)
          BALANCED // partition function
      ));

  registerMTreeConfig( "mMRad",
      MTreeConfig(
          80,      // maxNodeEntries
          mM_RAD,   // promote function (min. covering radius)
```

```
            BALANCED // partition function
        ));

    registerMTreeConfig( "mMRadHP",
        MTreeConfig(
            80,      // maxNodeEntries
            mM_RAD,   // promote function (min. covering radius)
            GENERALIZED_HYPERPLANE // partition function
        ));
    initialized = true;
}
```