X-Tree Algebra

Contents

1 Overview

This algebra provides the `xtree` type constructor and the following operators (`DATA` must be of the same type as the attributes indized in the x-tree or a hpoint):

- `_ creatextree [_]`
  Creates a new xtree from a relation or tuple stream.

  Signature: `relation/tuple-stream x <attr-name> -> xtree`
  Example: `let xt = strassen creatextree[geoData]`

- `_ creatextree2 [_, _]`
  Like creatextree, but additionaly allows to specify another than the default xtree config.

  Signature: `relation/tuple-stream x <attr-name> x <config-name> -> xtree`
  Example: `let xt = strassen creatextree2[geoData, limit80e]`

- `_ creatextree3 [_, _, _]`
  Like creatextree2, but additionaly allows to specify another than the default gethpoint or getbbox function (if a gethpoint and a getbbox function with the same name are defined, the gethpoint function will be used).

  Signature: `relation/tuple-stream x <attr-name> x <config-name> x <getdatafun-name> -> xtree`
  Example: `let xt = strassen creatextree3[geoData, limit80e, native]`

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

  Signature: `xtree relation x DATA x real -> tuple stream`
  Example: `xt strassen rangesearch[p, 1000]`

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

  Signature: `xtree relation x DATA x int -> tuple stream`
  Example: `xt strassen nnsearch[p, 5] count`

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

  Signature: `xtree relation x hrect -> tuple stream`
  Example: `xt strassen windowintersects[r] count`

- `_ _ nnscan [_]`
  Returns a tuple stream, which contains the ranking of the indized elements, based on the distance to the query point. The relation must contain at least the same tuples, that had been used

to create the xtree.

Signature: `xtree relation x DATA -> tuple stream`
Example: `xt strassen nnscan[p] head[ 10 ] count`

2 Headerfile `XTreeAlgebra.h`

January-May 2008, Mirko Dibbert

2.1 Overview

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

```
#ifndef __XTREE_ALGEBRA_H__
#define __XTREE_ALGEBRA_H__

#include "GeneralTreeAlgebra.h"

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


//////////////////////////////////////////////////////////////////////
// enables debugging mode for the xtree-algebra
//////////////////////////////////////////////////////////////////////
// #define __XTREE_DEBUG


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

```
#define __XTREE_PRINT_INSERT_INFO


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


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

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


//////////////////////////////////////////////////////////////////////
// use min deadspace to resolve ties instead of minimal area in the
// topologicalSplit and overlapMinimalSplit methods
//////////////////////////////////////////////////////////////////////
#define __XTREE_SPLIT_USE_MIN_DEADSPACE


namespace xtreeAlgebra
{

//////////////////////////////////////////////////////////////////////
// these constants are used within the XTree::split method
//////////////////////////////////////////////////////////////////////
const double MAX_OVERLAP = 0.2;
const double MIN_FANOUT  = 0.35;


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


//////////////////////////////////////////////////////////////////////
// intevall of printing statistic infos in the insert method
// (only used, if __XTREE_PRINT_INSERT_INFO is defined)
//////////////////////////////////////////////////////////////////////
const int insertInfoInterval = 10;
```

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

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


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


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


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


//////////////////////////////////////////////////////////////////////
// priorities of the defined node types
// (higher priorities result into a higher probablility for
// nodes of the respective type to remain in the node cache)
//////////////////////////////////////////////////////////////////////
const unsigned leafPrio       = 0; // default = 0
const unsigned internalPrio   = 1; // default = 1
const unsigned supernodePrio  = 2; // default = 2


//////////////////////////////////////////////////////////////////////
// constants for the node type id's
//////////////////////////////////////////////////////////////////////
const NodeTypeId LEAF      = 0;
const NodeTypeId INTERNAL  = 1;
const NodeTypeId SUPERNODE = 2;



// define __XTREE_ANALYSE_STATS if __XTREE_PRINT_SEARCH_INFO or
// __XTREE_PRINT_STATS_TO_FILE has been defined
#ifdef __XTREE_PRINT_SEARCH_INFO
  #define __XTREE_ANALYSE_STATS
#else
```

```
  #ifdef __XTREE_PRINT_STATS_TO_FILE
  #define __XTREE_ANALYSE_STATS
  #endif
#endif
} // namespace xtreeAlgebra

#endif // #ifndef __XTREE_ALGEBRA_H__
```

2.2 Headerfile `XTree.h`

January-May 2008, Mirko Dibbert

2.2.1 Overview

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

2.2.2 includes and defines

```
#ifndef __XTREE_H__
#define __XTREE_H__

#include "XTreeConfig.h"
#include "XTreeAlgebra.h"
#include "XTreeBase.h"

namespace xtreeAlgebra
{
```

2.2.3 Struct `SearchBestPathEntry`:

This struct is needed in the `xtree::insert` method.

```
struct SearchBestPathEntry
{
    SearchBestPathEntry(InternalEntry* _entry, unsigned _index)
```

```
        : entry(_entry), index(_index)
    {}

    InternalEntry *entry;
    unsigned index;
}; // struct SerachBestPathEntry
```

2.2.4 Struct `RemainingNodesEntryNNS`:

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

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

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

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

2.2.5 Struct `NNEntry`:

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

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

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

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

2.2.6 Struct `NNEntry`:

This struct is used for the nnscan methods.

```
    struct NNScanEntry
    {
        bool isNodeId;
        union
        {
            SmiRecordId nodeId;
            TupleId tid;
        };
        double dist;

        NNScanEntry(TupleId _tid, double _dist)
        : isNodeId(false), tid(_tid), dist(_dist)
        {}

        NNScanEntry(SmiRecordId _nodeId, double _dist)
        : isNodeId(true), nodeId(_nodeId), dist(_dist)
        {}

        bool operator > (const NNScanEntry& op2) const
        {
            if (dist > op2.dist)
                return true;
            else if (dist < op2.dist)
                return false;

            // dist == op2.dist
            if (isNodeId)
            {
                if (op2.isNodeId)
                { // isNodeId && op2.isNodeId
                    if (nodeId > op2.nodeId)
                        return true;
                    else
                        return false;
                }
                else
                { // isNodeId && !op2.isNodeId
                    return false;
                }
            }
            else
            { // !isNodeId
                if (op2.isNodeId)
                { // !isNodeId && op2.isNodeId
                    return true;
                }
                else
                { // !isNodeId && !op2.isNodeId
                    if (tid > op2.tid)
                        return true;
                    else
                        return false;
                }
            }
        }
    };
```

2.3 Struct `Header`

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

        unsigned supernodeCount;
        STRING_T configName;
        STRING_T typeName;
        int getdataType;
        STRING_T getdataName;
        unsigned dim;
        bool initialized;
    }; // struct Header
```

2.4 Class `XTree`

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

Default Constructor, creates a new x-tree.

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

Constructor, opens an existing tree.

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

Default copy constructor

```
inline XTree(const XTree &xtree)
    : gtree::Tree<Header>(xtree)
{
    if (xtree.isInitialized())
        initialize();
}
```

Destructor

```
inline ~XTree()
{}
```

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

```
void initialize(
    unsigned dim,
    const string &configName,
    const string &typeName,
    int getdataType,
    const string &getdataName);
```

Creates a new LeafEntry from `bbox` and inserts it into the xtree.

```
void insert(LeafEntry *entry);
```

Returns all entries, wich have a maximum (eucledean) distance of `rad` to the given point in the result list (for spatial data, the distance to the center of the bounding box is used).

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

Returns all entries, wich intersect the given hyper rectangle in the result list.

```
void windowIntersects(HRect *r, list<TupleId> *results);
```

Returns the `nncount` nearest neighbours ot the point in the result list.

```
void nnSearch(HPoint *p, int nncount, list<TupleId> *results);
```

These methods are used for the nnscan operator, which returns a ranking of the indized elements, based on their distance to to the reference object p.

```
void nnscan_init(HPoint *p);
TupleId nnscan_next();
void nnscan_cleanup();
```

Returns the count of all supernodes.

```
inline unsigned supernodeCount()
{ return header.supernodeCount;}
```

Returns the dimension of the assigned bounding boxes.

```
inline unsigned dim()
{ return header.dim; }
```

Returns the name of the used XTreeConfig object.

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

Returns the name of the assigned type constructor

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

Returns the type of the assigned getdata function (gethpoint or gethrect).

```
inline int getdataType()
{ return header.getdataType; }
```

Returns the name of the assigned getdata function.

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

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

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

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

```
void printTreeInfos()
{
    cmsg.info() << endl
        << "<xtree infos>" << endl
        << "   dimension                 : "
        << dim() << endl
        << "   entries                   : "
        << entryCount() << endl
```

```
                << "   height                   : "
                << height() << endl
                << "   directory nodes          : "
                << internalCount() << endl
                << "   leaf nodes               : "
                << leafCount() << endl
                << "   supernodes               : "
                << supernodeCount() << endl
                << "   assigned config          : "
                << configName() << endl
                << "   assigned type            : "
                << header.typeName << endl
                << "   assigned getdata function : "
                << header.getdataName << endl
                << endl << endl;
            cmsg.send();
        }

    private:
        XTreeConfig config; // assigned XTreeConfig object
```

Adds prototypes for the avaliable node types.

```
        void registerNodePrototypes();
```

Initializes `config` and calls the `registerNodePrototypes` method. This method needs an initialized header to work.

```
        void initialize();
```

Splits a node.

```
        void split();
```

Topological Split.

```
        template<class TEntry>
        unsigned topologicalSplit(
                vector<TEntry*> *in,
                vector<TEntry*> *out1,
                vector<TEntry*> *out2);
```

Overlap minimal split.

```
        unsigned overlapMinimalSplit(
                vector<InternalEntry*> *in,
                vector<InternalEntry*> *out1,
                vector<InternalEntry*> *out2);
```

Selects one of the chields of `treeMngr->curNode` as next node in the path.

```
        int chooseSubtree(HRect *bbox);

        vector<NNScanEntry> nnscan_queue;
        HPoint *nnscan_ref;
    }; // class XTree
```

2.5 Struct *SortedHRect*

Auxiliary structure for `topologicalSplit` and `overlapMinimalSplit`.

```
template <class TEntry>
struct SortedBBox
{
    unsigned dim;
    unsigned index;
    HRect *bbox;
    TEntry *entry;
```

Sort function for lower bound sort.

```
        static int sort_lb(const void *lhs, const void *rhs)
        {
            const SortedBBox<TEntry> *s1 =
                    static_cast<const SortedBBox<TEntry>*>(lhs);
            const SortedBBox<TEntry> *s2 =
                    static_cast<const SortedBBox<TEntry>*>(rhs);

            unsigned dim = s1->dim;
            double diff = s1->bbox->lb(dim) - s2->bbox->lb(dim);

            if (diff < 0.0)
                return -1;
            else if (diff == 0.0)
                return 0;
            else
                return 1;
        }
```

Sort function for upper bound sort.

```
        static int sort_ub(const void *lhs, const void *rhs)
        {
            const SortedBBox<TEntry> *s1 =
                    static_cast<const SortedBBox<TEntry>*>(lhs);
            const SortedBBox<TEntry> *s2 =
                    static_cast<const SortedBBox<TEntry>*>(rhs);

            unsigned dim = s1->dim;
            double diff = s1->bbox->ub(dim) - s2->bbox->ub(dim);

            if (diff < 0.0)
                return -1;
            else if (diff == 0.0)
                return 0;
            else
                return 1;
        }
    };
```

Method *topologicalSplit*:

```
template<class TEntry>
unsigned XTree::topologicalSplit(
        vector<TEntry*>* in,
        vector<TEntry*>* out1,
        vector<TEntry*>* out2)
{
    unsigned n = in->size();
    unsigned minEntries = static_cast<unsigned>(n * 0.4);
    if (minEntries == 0)
        minEntries = 1;

    HRect *bbox1_lb, *bbox2_lb, *bbox1_ub, *bbox2_ub;
    SortedBBox<TEntry> sorted_lb[n], sorted_ub[n];


    /////////////////////////////////////////////////////////////////
    // chose split axis
    /////////////////////////////////////////////////////////////////
    unsigned split_axis = 0;
    double marginSum;
    double minMarginSum = numeric_limits<double>::infinity();
    for(unsigned d = 0; d < header.dim; ++d)
    {
        // sort entries by lower/upper bound for actual dimension
        for (unsigned i = 0; i < n; ++i)
        {
            sorted_lb[i].dim = sorted_ub[i].dim = d;
            sorted_lb[i].index = sorted_ub[i].index = i;
            sorted_lb[i].bbox = sorted_ub[i].bbox = (*in)[i]->bbox();
        }
        qsort(sorted_lb, n, sizeof(SortedBBox<TEntry>),
                SortedBBox<TEntry>::sort_lb);
        qsort(sorted_ub, n, sizeof(SortedBBox<TEntry>),
                SortedBBox<TEntry>::sort_ub);

        for (unsigned k = 0; k < n - 2*minEntries + 1; ++k)
        { // for all possible distributions
            // compute bounding boxes for actual distribution
            unsigned pos = 0;

            bbox1_lb = new HRect(*(sorted_lb[pos].bbox));
            bbox1_ub = new HRect(*(sorted_ub[pos].bbox));
            ++pos;
            while(pos < minEntries+k)
            {
                bbox1_lb->unite(sorted_lb[pos].bbox);
                bbox1_ub->unite(sorted_ub[pos].bbox);
                ++pos;
            }

            bbox2_lb = new HRect(*(sorted_lb[pos].bbox));
            bbox2_ub = new HRect(*(sorted_ub[pos].bbox));
            ++pos;
            while(pos < n)
            {
                bbox2_lb->unite(sorted_lb[pos].bbox);
                bbox2_ub->unite(sorted_ub[pos].bbox);
                ++pos;
            }

            // compute marginSum
            marginSum = bbox1_lb->margin();
            marginSum += bbox1_ub->margin();
            marginSum += bbox2_lb->margin();
            marginSum += bbox2_ub->margin();

            if (marginSum < minMarginSum)
            {
                minMarginSum = marginSum;
                split_axis = d;
            }

            delete bbox1_lb;
            delete bbox1_ub;
            delete bbox2_lb;
            delete bbox2_ub;
        }

    }

    /////////////////////////////////////////////////////////////////
    // chose split index
    /////////////////////////////////////////////////////////////////
    unsigned split_index = minEntries;
    double overlap;
    double minOverlap = numeric_limits<double>::infinity();
    bool lb;

    #ifdef XTREE_SPLIT_USE_MIN_DEADSPACE
    double minDeadspace = numeric_limits<double>::infinity();
    #else
    double area;
    double minArea = numeric_limits<double>::infinity();
    #endif

    // sort entries by lower/upper bound for actual dimension
    for (unsigned i = 0; i < n; ++i)
    {
        sorted_lb[i].dim = sorted_ub[i].dim = split_axis;
        sorted_lb[i].index = sorted_ub[i].index = i;
        sorted_lb[i].bbox = sorted_ub[i].bbox = (*in)[i]->bbox();
        sorted_lb[i].entry = sorted_ub[i].entry = (*in)[i];
    }
    qsort(sorted_lb, n, sizeof(SortedBBox<TEntry>),
                    SortedBBox<TEntry>::sort_lb);
            qsort(sorted_ub, n, sizeof(SortedBBox<TEntry>),
                    SortedBBox<TEntry>::sort_ub);

            #ifdef XTREE_SPLIT_USE_MIN_DEADSPACE
            double deadspace_lb = 0.0;
            double deadspace_ub = 0.0;
            #endif

            for (unsigned k = 0; k < n - 2*minEntries + 1; ++k)
            { // for all possible distributions
                // compute bounding boxes for actual distribution
                unsigned pos = 0;

                bbox1_lb = new HRect(*(sorted_lb[pos].bbox));
                bbox1_ub = new HRect(*(sorted_ub[pos].bbox));
                #ifdef XTREE_SPLIT_USE_MIN_DEADSPACE
                deadspace_lb -= sorted_lb[pos].bbox->area();
                deadspace_ub -= sorted_ub[pos].bbox->area();
                #endif
                ++pos;
                while(pos < minEntries+k)
                {
                    bbox1_lb->unite(sorted_lb[pos].bbox);
                    bbox1_ub->unite(sorted_ub[pos].bbox);
                    #ifdef XTREE_SPLIT_USE_MIN_DEADSPACE
                    deadspace_lb -= sorted_lb[pos].bbox->area();
                    deadspace_ub -= sorted_ub[pos].bbox->area();
                    #endif
                    ++pos;
                }

                bbox2_lb = new HRect(*(sorted_lb[pos].bbox));
                bbox2_ub = new HRect(*(sorted_ub[pos].bbox));
                #ifdef XTREE_SPLIT_USE_MIN_DEADSPACE
                deadspace_lb -= sorted_lb[pos].bbox->area();
                deadspace_ub -= sorted_ub[pos].bbox->area();
                #endif
                ++pos;
                while(pos < n)
                {
                    bbox2_lb->unite(sorted_lb[pos].bbox);
                    bbox2_ub->unite(sorted_ub[pos].bbox);
                    #ifdef XTREE_SPLIT_USE_MIN_DEADSPACE
                    deadspace_lb -= sorted_lb[pos].bbox->area();
                    deadspace_ub -= sorted_ub[pos].bbox->area();
                    #endif
                    ++pos;
                }

        #ifdef XTREE_SPLIT_USE_MIN_DEADSPACE
        // use minimal deadspace to resolve ties
                // compute overlap and area of lb_sort
                overlap = bbox1_lb->overlap(bbox2_lb);
                deadspace_lb += bbox1_lb->area() + bbox2_lb->area();
                if ((overlap < minOverlap) ||
                   ((overlap == minOverlap) && (deadspace_lb < minDeadspace)))
                {
                    minOverlap = overlap;
                    minDeadspace = deadspace_lb;
                    split_index = minEntries+k;
                    lb = true;
                }

                // compute overlap and area of ub_sort
                overlap = bbox1_ub->overlap(bbox2_ub);
                deadspace_ub += bbox1_ub->area() + bbox2_ub->area();
                if ((overlap < minOverlap) ||
                   ((overlap == minOverlap) && (deadspace_ub < minDeadspace)))
                {
                    minOverlap = overlap;
                    minDeadspace = deadspace_ub;
                    split_index = minEntries+k;
                    lb = false;
                }
        #else
        // use minimal area to resolve ties
                overlap = bbox1_ub->overlap(bbox2_ub);
                area = bbox1_lb->area() + bbox2_lb->area();
                if ((overlap < minOverlap) ||
                        ((overlap == minOverlap) && (area < minArea)))
                {
                    minOverlap = overlap;
                    minArea = area;
                    split_index = minEntries+k;
                    lb = true;
                }

                // compute overlap and area of ub_sort
                overlap = bbox1_ub->overlap(bbox2_ub);
                area = bbox1_ub->area() + bbox2_ub->area();
                if ((overlap < minOverlap) ||
                        ((overlap == minOverlap) && (area < minArea)))
                {
                    minOverlap = overlap;
                    minArea = area;
                    split_index = minEntries+k;
                    lb = false;
                }
        #endif

                delete bbox1_lb;
                delete bbox1_ub;
                delete bbox2_lb;
                delete bbox2_ub;
            }
            // compute distribution
            if (lb)
            {
                unsigned i;
                for (i = 0; i < split_index; ++i)
                    out1->push_back(sorted_lb[i].entry);
                for (; i < n; ++i)
                    out2->push_back(sorted_lb[i].entry);
            }
            else
            {
                unsigned i;
                for (i = 0; i < split_index; ++i)
                    out1->push_back(sorted_ub[i].entry);
                for (; i < n; ++i)
                    out2->push_back(sorted_ub[i].entry);
            }
            return split_axis;
         }

        } // namespace xtreeAlgebra
        #endif // #ifndef __XTREE_H__
```

3 Headerfile `XTreeBase.h`

January-May 2008, Mirko Dibbert

3.1 Overview

This file contains the declarations of the xtree nodes and node entries.

3.2 includes and defines

```
#ifndef __XTREE_BASE_H__
#define __XTREE_BASE_H__

#include "RelationAlgebra.h"
#include "XTreeAlgebra.h"

namespace xtreeAlgebra {

using gtree::NodePtr;
```

3.3 Class `SplitHist`

This class implements the split history for internal entries, which is stored in one bit per dimension.

```
class SplitHist
{
public:
```

Contstructor (creates an empty split history)

```
        inline SplitHist(unsigned dim)
            : m_len((dim/8)+1), m_hist(new char[m_len])
        { memset(m_hist, 0, m_len); }
```

Default copy constructor.

```
        inline SplitHist(const SplitHist &e)
             : m_len(e.m_len), m_hist(new char[m_len])
        { memcpy(m_hist, e.m_hist, m_len); }
```

Constructor (reads the histoory from buffer and increases offset)

```
        inline SplitHist(const char *const buffer, int &offset)
        { read(buffer, offset); }
```

Destructor.

```
        inline ~SplitHist()
        { delete[] m_hist; }
```

Sets bit n in the Split history.

```
        inline void set(unsigned n)
        {
            #ifdef __XTREE_DEBUG
            assert(n < 8*m_len);
            #endif

            unsigned i = n/8;
            m_hist[i] |= (1 << (n - 8*i));
        }
```

Logical OR operator.

```
        inline void operator |= (const SplitHist &rhs)
        {
            for (unsigned i=0; i<m_len; ++i)
                m_hist[i] |= rhs.m_hist[i];
        }
```

Logical AND operator.

```
        inline void operator &= (const SplitHist &rhs)
        {
            for (unsigned i=0; i<m_len; ++i)
                m_hist[i] &= rhs.m_hist[i];
        }
```

Access operator (returns `true` if bit n is set)

```
        inline bool operator [] (unsigned n)
        {
            #ifdef __XTREE_DEBUG
            assert(n < 8*m_len);
            #endif

            int i = static_cast<int>(n/8);
            return (m_hist[i] & (1 << (n - 8*i)));
        }
```

Writes the history to buffer and increases offset.

```
        inline void write(char *const buffer, int &offset) const
        {
            memcpy(buffer+offset, &m_len, sizeof(unsigned));
            offset += sizeof(unsigned);

            memcpy(buffer+offset, m_hist, m_len);
            offset += m_len;
        }
```

Returns the size of the history in bytes.

```
        inline size_t size()
        { return sizeof(unsigned) + m_len*sizeof(char); }

    private:
```

Reads the history from buffer and increases offset.

```
        inline void read(const char* const buffer, int &offset)
        {
            memcpy(&m_len, buffer+offset, sizeof(unsigned));
            offset += sizeof(unsigned);

            m_hist = new char[m_len];
            memcpy(m_hist, buffer+offset, m_len);
            offset += m_len;
        }

        unsigned m_len;
        char *m_hist;
    };
```

3.4 Class *LeafEntry*

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

    public:
```

Default constructor.

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

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

```
        inline LeafEntry(TupleId tid, HPoint *p)
            : m_tid(tid), m_isPoint(true), m_bbox(p->bbox()), m_point(p)
        {
            #ifdef __XTREE_DEBUG
            assert(p);
            #endif
        }
```

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

```
        inline LeafEntry(TupleId tid, HRect *bbox)
            : m_tid(tid), m_isPoint(false), m_bbox(bbox), m_point(0)
        {
            #ifdef __XTREE_DEBUG
            assert(m_bbox);
            #endif
        }
```

Default copy constructor.

```
        inline LeafEntry(const LeafEntry &e)
            : m_tid(e.m_tid), m_isPoint(e.m_isPoint),
              m_bbox(e.m_bbox), m_point(e.m_point)
        {}
```

Destructor.

```
        inline ~LeafEntry()
        {
            delete m_bbox;
            if (m_point)
                delete m_point;
        }
```

Returns the tuple id of the entry.

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

Returns a reference to the `HRect` object.

```
        inline HRect *bbox()
        { return m_bbox; }
```

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

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

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

    // read m_isPoint
    memcpy(buffer+offset, &m_isPoint, sizeof(bool));
    offset += sizeof(bool);

    if (m_isPoint)
        m_point->write(buffer, offset);
    else
        m_bbox->write(buffer, offset);
}
```

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

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

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

    // read m_isPoint
    memcpy(&m_isPoint, buffer+offset, sizeof(bool));
    offset += sizeof(bool);

    if (m_isPoint)
    {
        m_point = new HPoint(buffer, offset);
        m_bbox = m_point->bbox();
    }
    else
    {
        m_point = 0;
        m_bbox = new HRect(buffer, offset);
    }
}
```

Returns the size of the entry on disc.

```
inline size_t size()
{
```

```
            if (m_isPoint)
            {
                return gtree::LeafEntry::size() +
                        sizeof(TupleId) + sizeof(bool) + m_point->size();
            }
            else
            {
                return gtree::LeafEntry::size() + sizeof(bool) +
                        sizeof(TupleId) + sizeof(bool) + m_bbox->size();
            }
        }
```

Returns the square of the Euclidean distance between `p` and the entry data (if the data is no point, the distance to the center of the bounding box is returned).

```
        double dist(HPoint *p)
        {
            if (m_isPoint)
                return SpatialDistfuns::euclDist2(p, m_point);
            else
            {
                HPoint c = m_bbox->center();
                return SpatialDistfuns::euclDist2(p, &c);
            }
        }


    private:
        TupleId   m_tid;      // tuple-id of the entry
        bool      m_isPoint;  // true, if data is point data
        HRect     *m_bbox;    // bounding box of the entry
        HPoint    *m_point;   // used for point data
    }; // class LeafEntry
```

3.5 Class *InternalEntry*

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

Default constructor.

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

Constructor.

```
        inline InternalEntry(HRect *bbox, SmiRecordId _chield)
            : gtree::InternalEntry(_chield), m_bbox(bbox),
              m_history(new SplitHist(m_bbox->dim()))
        {}
```

Constructor.

```
inline InternalEntry(
        HRect *bbox, SmiRecordId _chield, SplitHist *_hist)
    : gtree::InternalEntry(_chield), m_bbox(bbox),
      m_history(new SplitHist(*_hist))
{}
```

Default copy constructor.

```
inline InternalEntry(const InternalEntry &e)
    : gtree::InternalEntry(e), m_bbox(new HRect(*e.m_bbox)),
      m_history(e.history())
{}
```

Destructor.

```
inline ~InternalEntry()
{
    delete m_bbox;
    delete m_history;
}
```

Returns a reference to the `HRect` object.

```
inline HRect *bbox()
{ return m_bbox; }
```

Replaces the bounding box with new one (used during split).

```
inline void replaceHRect(HRect* _bbox)
{
    delete m_bbox;
    m_bbox = _bbox;

    #ifdef __XTREE_DEBUG
    assert(m_bbox);
    #endif
}
```

Returns the split history.

```
inline SplitHist *history() const
{ return m_history; }
```

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

```
inline void write(char *const buffer, int &offset) const
{
    gtree::InternalEntry::write(buffer, offset);
    m_bbox->write(buffer, offset);
    m_history->write(buffer, offset);
}
```

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

```
        inline void read(const char *const buffer, int &offset)
        {
            gtree::InternalEntry::read(buffer, offset);
            m_bbox = new HRect(buffer, offset);
            m_history = new SplitHist(buffer, offset);
        }
```

Returns the size of the entry on disc.

```
        inline size_t size()
        {
            return gtree::InternalEntry::size() +
                m_bbox->size() + m_history->size();
        }

    private:
        HRect     *m_bbox;    // bounding box of the entry
        SplitHist *m_history; // split history
    }; // class DirEntry
```

3.6 Class *LeafNode*

```
    class LeafNode
            : public gtree::LeafNode<LeafEntry>
    {
    public:
```

Default constructor.

```
        inline LeafNode(gtree::NodeConfigPtr config)
            : gtree::LeafNode<LeafEntry>(config, 0)
        {}
```

Default copy constructor.

```
        inline LeafNode(const LeafNode& node)
            : gtree::LeafNode<LeafEntry>(node)
        {}
```

Virtual destructor.

```
        inline virtual ~LeafNode()
        {}
```

Returns a reference to a copy of the node.

```
        virtual LeafNode *clone() const
        { return new LeafNode(*this); }
```

Returns the union of all contained bounding boxes.

```
        inline HRect *bbox()
        {
            iterator iter = begin();
            HRect *new_bbox = new HRect(*(*iter)->bbox());
            while(++iter != end())
                new_bbox->unite((*iter)->bbox());

            return new_bbox;
        }
```

Returns the union of the given entry vector.

```
        static HRect *bbox(vector<LeafEntry*> *entries)
        {
            iterator iter = entries->begin();
            HRect *new_bbox = new HRect(*(*iter)->bbox());
            while(++iter != entries->end())
                new_bbox->unite((*iter)->bbox());

            return new_bbox;
        }
    }; // class "LeafNode"[4]
```

3.7 Class *InternalNode*

```
    class InternalNode
            : public gtree::InternalNode<InternalEntry>
    {
    public:
```

Default constructor.

```
        inline InternalNode(
                gtree::NodeConfigPtr config,
                gtree::NodeConfigPtr defaultConfig,
                gtree::NodeConfigPtr supernodeConfig)
        : gtree::InternalNode<InternalEntry>(config, 0),
          m_defaultConfig(defaultConfig),
          m_supernodeConfig(supernodeConfig)
        {}
```

Default copy constructor.

```
        inline InternalNode(const InternalNode &node)
        : gtree::InternalNode<InternalEntry>(node),
          m_defaultConfig(node.m_defaultConfig),
          m_supernodeConfig(node.m_supernodeConfig)
        {}
```

MemSize (used to update used node cache size).

```
    unsigned memSize(bool recompute) const
    {
        return gtree::InternalNode<InternalEntry>::
            memSize(recompute) + 2*sizeof(gtree::NodeConfigPtr);
    }
```

Virtual destructor.

```
    inline virtual ~InternalNode()
    {}
```

Returns a reference to a copy of the node.

```
    virtual InternalNode *clone() const
    { return new InternalNode(*this); }
```

Sets supernode state.

```
    inline void setSupernode()
    { m_config = m_supernodeConfig; }
```

Resets supernode state.

```
    inline void resetSupernode()
    { m_config = m_defaultConfig; }
```

Returns `true` if `this` is a supernode.

```
    inline bool isSupernode()
    { return this->m_config->type() == SUPERNODE; }
```

Returns the union of all contained bounding boxes.

```
    inline HRect *bbox()
    {
        iterator iter = begin();
        HRect *new_bbox = new HRect(*(*iter)->bbox());
        while(++iter != end())
            new_bbox->unite((*iter)->bbox());

        return new_bbox;
    }
```

Returns the union of the given entry vector.

```
    static HRect *bbox(vector <InternalEntry*> *entries)
    {
        iterator iter = entries->begin();
        HRect *new_bbox = new HRect(*(*iter)->bbox());
        while(++iter != entries->end())
            new_bbox->unite((*iter)->bbox());
        return new_bbox;
    }

private:
```

This config objects are needed to change a node into a supernode and vice versa.

```
    gtree::NodeConfigPtr m_defaultConfig;
    gtree::NodeConfigPtr m_supernodeConfig;
}; // class "InternalNode"[4]
```

3.8 Typedefs

```
typedef LeafNode* LeafNodePtr;
typedef InternalNode* InternalNodePtr;

} // namespace xtreeAlgebra
#endif // #ifndef __XTREE_BASE_H__
```

4 Headerfile `XTreeConfig.h`

January-May 2008, Mirko Dibbert

4.1 Overview

This headerfile contains the `XTreeConfigReg` class, which provides a set of configurations. Each configuration is identified with a unique name and sets the min/max count of entries and max count of pages per node.

All avaliable config objects are defined in the initialize function (file `MTreeConfig.cpp`) and could be selected with the creatextree2 operator (creatextree uses the default values).

4.2 Includes and defines

```
#ifndef XTREE_CONFIG_H
#define XTREE_CONFIG_H

#include "XTreeAlgebra.h"

namespace xtreeAlgebra
{


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

4.3 Struct *XTreeConfig*

```
struct XTreeConfig
{
```

Config objects for all node types.

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

Constructor (creates object with default values).

```
    XTreeConfig()
        : leafNodeConfig(
                LEAF, leafPrio, minLeafEntries,
                maxLeafEntries, maxLeafPages, leafCacheable),
          internalNodeConfig(
                INTERNAL, internalPrio, minIntEntries,
                maxIntEntries, maxIntPages, internalCacheable)
  {}
```

Constructor (creates objects with the given parameters).

```
    XTreeConfig(
            NodeConfig _leafNodeConfig,
            NodeConfig _internalNodeConfig)
        : leafNodeConfig(_leafNodeConfig),
          internalNodeConfig(_internalNodeConfig)
  {}
}; // struct XTreeConfig
```

4.4 Class `XTreeConfigReg`

```
class XTreeConfigReg
{

public:
```

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

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

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

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

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

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

Registeres all config objects.

```
        static void initialize();

    private:
        static map<string, XTreeConfig> configs;
        static bool initialized;
    }; // class XTreeConfigReg

    } // namespace xtreeAlgebra

    #endif
```