BTree

3

Generated by Doxygen 1.8.6

Sat Dec 5 2015 02:27:43

Contents

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 BM_BufferPool Struct Reference

```
#include <buffer_mgr.h>
```

Public Attributes

- char ∗ **pageFile**
- int **numPages**
- **ReplacementStrategy strategy**
- void ∗ **mgmtData**

3.1.1 Detailed Description

BM_BufferPool (p. 5) creates an in memory pool of pages which are read using storage manager.

The addition and deletion of pages in the pool is done based on the value of "strategy" All the pages are stored in mgmt data using a **BM_MetaList** (p. 6) struct object.

Definition at line 36 of file buffer_mgr.h.

3.1.2 Member Data Documentation

3.1.2.1 void∗ BM_BufferPool::mgmtData

use this one to store the bookkeeping info your buffer manager needs for a buffer pool

Definition at line 45 of file buffer_mgr.h.

3.1.2.2 int BM_BufferPool::numPages

contains the total number of pages currently in the pool

Definition at line 41 of file buffer_mgr.h.

3.1.2.3 char∗ BM_BufferPool::pageFile

contains the name of the database file

Definition at line 39 of file buffer_mgr.h.

3.1.2.4 ReplacementStrategy BM_BufferPool::strategy

specifies the order of addition and deletion of pages

Definition at line 43 of file buffer_mgr.h.

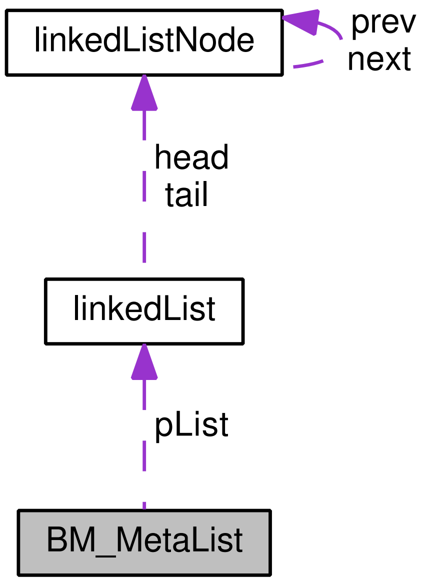
The documentation for this struct was generated from the following file:

- **buffer_mgr.h**

3.2 BM_MetaList Struct Reference

```
#include <buffer_mgr.h>
```

Collaboration diagram for BM_MetaList:



Public Attributes

- int **numReadIO**
- int **numWriteIO**
- **linkedList** ∗ **pList**

3.2.1 Detailed Description

This structure contains some book keeping information for the buffer pool, along with the pages in the pool.

Definition at line 79 of file buffer_mgr.h.

3.2.2 Member Data Documentation

3.2.2.1 int BM_MetaList::numReadIO

This variable maintains a count of Read IO since the time a buffer pool initiated.

Definition at line 82 of file buffer_mgr.h.

3.2.2.2 int BM_MetaList::numWriteIO

This variable maintains a count of Write IO since the time buffer pool is initiated.

Definition at line 84 of file buffer_mgr.h.

3.2.2.3 linkedList∗ BM_MetaList::pList

This variable is actually a linked list of pages in the pool, the data in the node of this linked list is of type **BM_Meta-Page** (p. 7).

Definition at line 86 of file buffer_mgr.h.

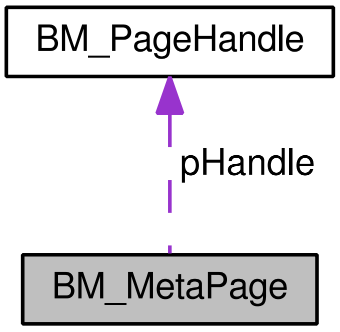
The documentation for this struct was generated from the following file:

- **buffer_mgr.h**

3.3 BM_MetaPage Struct Reference

```
#include <buffer_mgr.h>
```

Collaboration diagram for BM_MetaPage:



Public Attributes

- int **dirtyFlag**
- int **fixCount**
- **BM_PageHandle** ∗ **pHandle**

3.3.1 Detailed Description

This structures combines some basic meta information to a pageHandle, that allows us to maintain some book-keeping information for a given pageHandle.

Definition at line 65 of file buffer_mgr.h.

3.3.2 Member Data Documentation

3.3.2.1 int BM_MetaPage::dirtyFlag

This variable takes a value of 1 when a page is marked dirty, and it has a 0 value otherwise.

Definition at line 68 of file buffer_mgr.h.

3.3.2.2 int BM_MetaPage::fixCount

This variable maitains a count of the number times this page has been pinned (and still not unpinned).

Definition at line 70 of file buffer_mgr.h.

3.3.2.3 BM_PageHandle∗ BM_MetaPage::pHandle

This contains the main data for the page.

Definition at line 72 of file buffer_mgr.h.

The documentation for this struct was generated from the following file:

- **buffer_mgr.h**

3.4 BM_PageHandle Struct Reference

```
#include <buffer_mgr.h>
```

Public Attributes

- **PageNumber pageNum**
- char ∗ **data**

3.4.1 Detailed Description

BM_PageHandle (p. 8) budles a PageNumber (int) pageNum field along with its page contents in field "data".

Definition at line 52 of file buffer_mgr.h.

3.4.2 Member Data Documentation

3.4.2.1 char∗ BM_PageHandle::data

contains data corresponding to page number pageNum

Definition at line 57 of file buffer_mgr.h.

3.4.2.2 PageNumber BM_PageHandle::pageNum

denotes the page number for which the contents are stored in "data"

Definition at line 55 of file buffer_mgr.h.

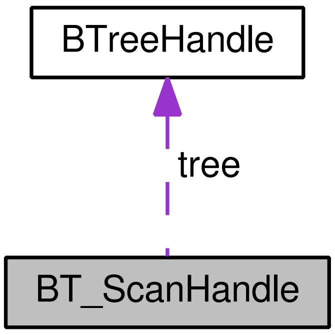
The documentation for this struct was generated from the following file:

- **buffer_mgr.h**

3.5 BT_ScanHandle Struct Reference

```
#include <btree_mgr.h>
```

Collaboration diagram for BT_ScanHandle:



Public Attributes

- **BTreeHandle** ∗ **tree**
- void ∗ **mgmtData**

3.5.1 Detailed Description

This structure stores the details of a tree scan operation.

Definition at line 70 of file btree_mgr.h.

3.5.2 Member Data Documentation

3.5.2.1 void∗ BT_ScanHandle::mgmtData

Contains meta data related to scan.

The meta data is stored using structure **BT_ScanMeta** (p. 9)

Definition at line 75 of file btree_mgr.h.

3.5.2.2 BTreeHandle∗ BT_ScanHandle::tree

The BTree to be scanned.

Definition at line 73 of file btree_mgr.h.

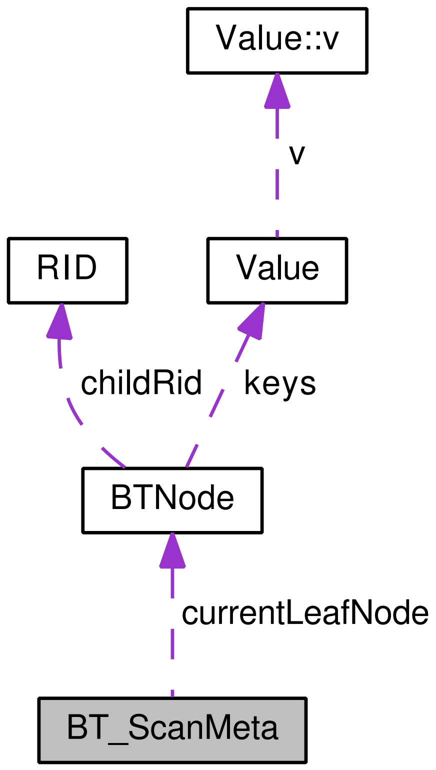
The documentation for this struct was generated from the following file:

- **btree_mgr.h**

3.6 BT_ScanMeta Struct Reference

```
#include <btree_mgr.h>
```

Collaboration diagram for BT_ScanMeta:



Public Attributes

- int **firstLeafPage**
- int **currentLeafPage**
- int **currentEntry**
- int **currentLeafEntry**
- **BTNode** ∗ **currentLeafNode**

3.6.1 Detailed Description

This structure stores an intermediate stage of the scan operation.

It is initialized to represent the first element in BTree (sorted ascending by key).

Definition at line 82 of file btree_mgr.h.

3.6.2 Member Data Documentation

3.6.2.1 int BT_ScanMeta::currentEntry

Contains the total number of entries scanned so far.

Definition at line 89 of file btree_mgr.h.

3.6.2.2 int BT_ScanMeta::currentLeafEntry

Contains the number of entries scanned so far in the currentLeafNode.

Definition at line 91 of file btree_mgr.h.

3.6.2.3 BTNode∗ BT_ScanMeta::currentLeafNode

Pointer to current leaf node being scanned.

This is initialized to firstLeafNode at the initiation of scan

Definition at line 93 of file btree_mgr.h.

3.6.2.4 int BT_ScanMeta::currentLeafPage

Page number of current lead node being scanned.

Definition at line 87 of file btree_mgr.h.

3.6.2.5 int BT_ScanMeta::firstLeafPage

Page number of first leaf node.

Definition at line 85 of file btree_mgr.h.

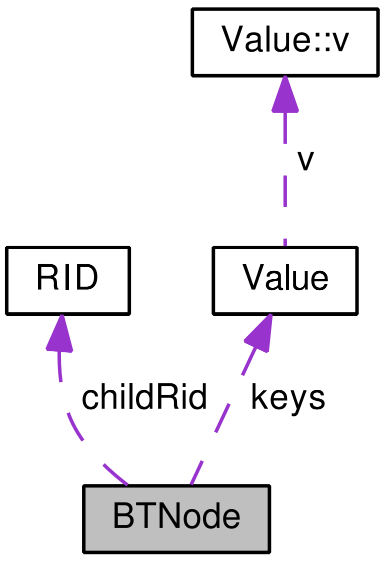
The documentation for this struct was generated from the following file:

- **btree_mgr.h**

3.7 BTNode Struct Reference

```
#include <btree_mgr.h>
```

Collaboration diagram for BTNode:



Public Attributes

- int **nodeNum**
- int **parent**
- int **numKeys**
- **bool isLeaf**
- **Value** ∗∗ **keys**

- int ∗ **childPage**
- **RID** ∗ **childRid**
- int **next**

3.7.1 Detailed Description

This structure represents an individual node of BTree.

Definition at line 25 of file btree_mgr.h.

3.7.2 Member Data Documentation

3.7.2.1 int∗ BTNode::childPage

This variable contains array of child pages.

Applicable only of internal nodes.

Definition at line 38 of file btree_mgr.h.

3.7.2.2 RID∗ BTNode::childRid

This variable contains array of child recordIds .

Applicable only for leaf nodes

Definition at line 40 of file btree_mgr.h.

3.7.2.3 bool BTNode::isLeaf

Boolean value.

If true, this node is leaf, else node is an internal node

Definition at line 34 of file btree_mgr.h.

3.7.2.4 Value∗∗ BTNode::keys

This variable contains array of keys.

Definition at line 36 of file btree_mgr.h.

3.7.2.5 int BTNode::next

Page number of the next node.

Definition at line 42 of file btree_mgr.h.

3.7.2.6 int BTNode::nodeNum

The page number (on disk) of this node.

Definition at line 28 of file btree_mgr.h.

3.7.2.7 int BTNode::numKeys

Number of keys in this node.

Definition at line 32 of file btree_mgr.h.

3.7.2.8 int BTNode::parent

The page number of parent node.

Definition at line 30 of file btree_mgr.h.

The documentation for this struct was generated from the following file:

- **btree_mgr.h**

3.8 BTreeHandle Struct Reference

```
#include <btree_mgr.h>
```

Public Attributes

- **DataType keyType**
- char ∗ **idxId**
- void ∗ **mgmtData**

3.8.1 Detailed Description

This structure stores a btree with name idxId.

The btree related data is stored in mgmtData.

Definition at line 11 of file btree_mgr.h.

3.8.2 Member Data Documentation

3.8.2.1 char∗ BTreeHandle::idxId

Name of the btree index.

Definition at line 16 of file btree_mgr.h.

3.8.2.2 DataType BTreeHandle::keyType

The data type of the btree key.

Currently only int values are supported.

Definition at line 14 of file btree_mgr.h.

3.8.2.3 void∗ BTreeHandle::mgmtData

Stores Btree related data, **BTreeMeta** (p. 14).

Definition at line 18 of file btree_mgr.h.

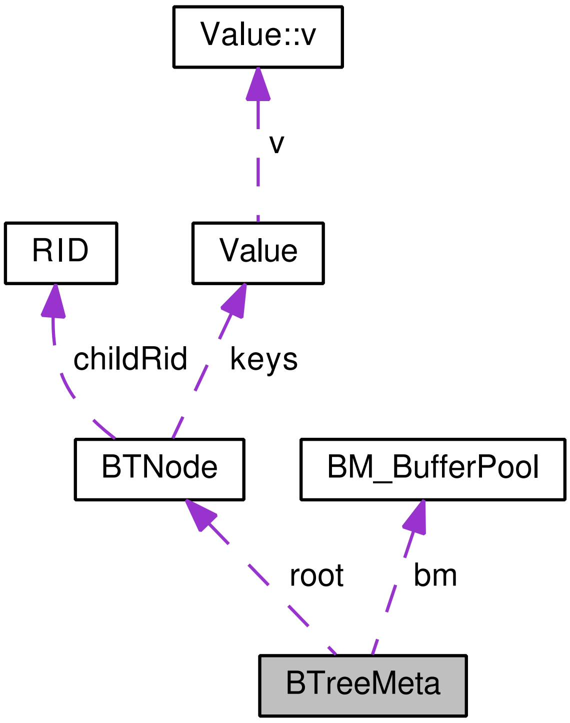
The documentation for this struct was generated from the following file:

- **btree_mgr.h**

3.9 BTreeMeta Struct Reference

```
#include <btree_mgr.h>
```

Collaboration diagram for BTreeMeta:



Public Attributes

- int **numEntries**
- int **numNodes**
- int **fanOut**
- **BTNode** ∗ **root**
- **BM_BufferPool** ∗ **bm**
- int **maxPageNum**

3.9.1 Detailed Description

This structure stores the details of a BTree.

It is stored in BTreeHandle->mgmtData

Definition at line 49 of file btree_mgr.h.

3.9.2 Member Data Documentation

3.9.2.1 BM_BufferPool∗ BTreeMeta::bm

Pointer to Buffer manager, to be used of accessing node pages.

Definition at line 60 of file btree_mgr.h.

3.9.2.2 int BTreeMeta::fanOut

The fanOut of the BTree (Max number of keys that a node can have)

Definition at line 56 of file btree_mgr.h.

3.9.2.3 int BTreeMeta::maxPageNum

The maximum value of any page number in a btree index structure.

Definition at line 62 of file btree_mgr.h.

3.9.2.4 int BTreeMeta::numEntries

Number of RIDs in BTree.

Definition at line 52 of file btree_mgr.h.

3.9.2.5 int BTreeMeta::numNodes

Number of Nodes in BTree.

Definition at line 54 of file btree_mgr.h.

3.9.2.6 BTNode∗ BTreeMeta::root

Pointer to Root Node.

Definition at line 58 of file btree_mgr.h.

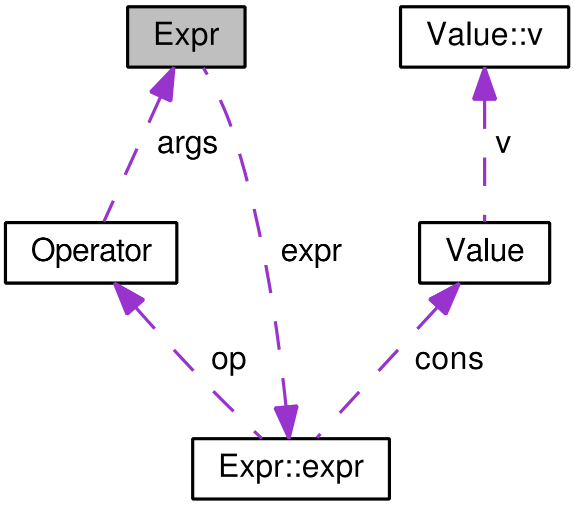
The documentation for this struct was generated from the following file:

- **btree_mgr.h**

3.10 Expr Struct Reference

```
#include <expr.h>
```

Collaboration diagram for Expr:



Classes

- union **expr**

Public Attributes

- **ExprType type**
- union **Expr::expr expr**

3.10.1 Detailed Description

Definition at line 14 of file expr.h.

3.10.2 Member Data Documentation

3.10.2.1 union Expr::expr Expr::expr

3.10.2.2 ExprType Expr::type

Definition at line 15 of file expr.h.

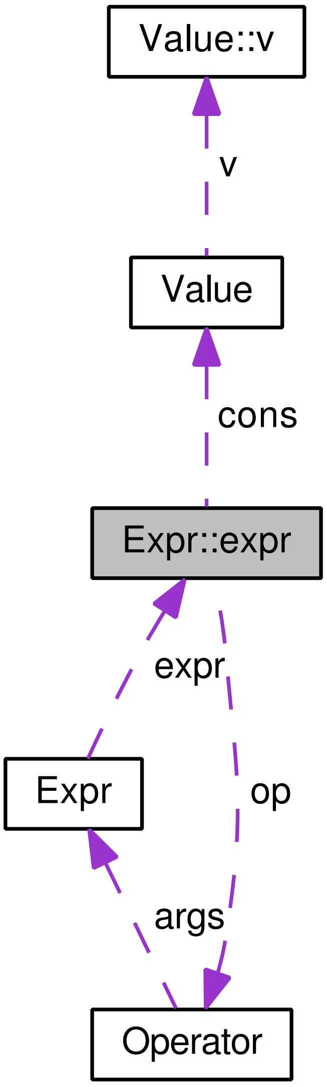
The documentation for this struct was generated from the following file:

- **expr.h**

3.11 Expr::expr Union Reference

```
#include <expr.h>
```

Collaboration diagram for Expr::expr:



Public Attributes

- **Value** ∗ **cons**
- int **attrRef**
- struct **Operator** ∗ **op**

3.11.1 Detailed Description

Definition at line 16 of file expr.h.

3.11.2 Member Data Documentation

3.11.2.1 int Expr::expr::attrRef

Definition at line 18 of file expr.h.

3.11.2.2 Value∗ Expr::expr::cons

Definition at line 17 of file expr.h.

3.11.2.3 struct Operator∗ Expr::expr::op

Definition at line 19 of file expr.h.

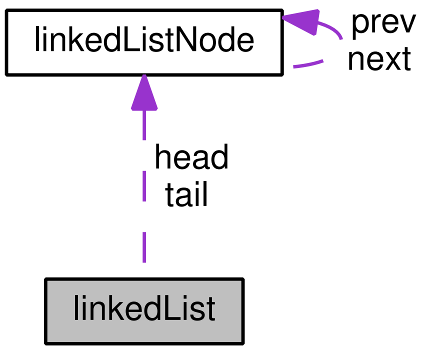
The documentation for this union was generated from the following file:

- **expr.h**

3.12 linkedList Struct Reference

```
#include <list.h>
```

Collaboration diagram for linkedList:



Public Attributes

- int **count**
- **linkedListNode** ∗ **head**
- **linkedListNode** ∗ **tail**

3.12.1 Detailed Description

This is a doubly linked list for storing pages in the buffer pool.

Doubly linked list allows me the flexibility to add and remove at both ends. This allows me to do basic operations like fifo additions in constant time (independent of the length of the linked list).

Definition at line 28 of file list.h.

3.12.2 Member Data Documentation

3.12.2.1 int linkedList::count

Definition at line 30 of file list.h.

3.12.2.2 linkedListNode∗ linkedList::head

The head pointer of the doubly linked list.

Definition at line 32 of file list.h.

3.12.2.3 linkedListNode∗ linkedList::tail

The tail pointer of the doubly linked list.

Definition at line 34 of file list.h.

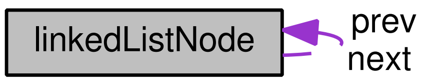
The documentation for this struct was generated from the following file:

- **list.h**

3.13 linkedListNode Struct Reference

`#include <list.h>`

Collaboration diagram for linkedListNode:



Public Attributes

- struct **linkedListNode** ∗ **next**
- struct **linkedListNode** ∗ **prev**
- void ∗ **value**

3.13.1 Detailed Description

This is a node for a doubly linked list **linkedList** (p. 18).

it contains its data as (void∗).

Definition at line 13 of file list.h.

3.13.2 Member Data Documentation

3.13.2.1 struct linkedListNode∗ linkedListNode::next

The next pointer of the linked list.

Definition at line 16 of file list.h.

3.13.2.2 struct linkedListNode∗ linkedListNode::prev

The previous pointer of the linked list.

Definition at line 18 of file list.h.

3.13.2.3 void∗ linkedListNode::value

The data cotained in this node.

When used in **linkedList** (p. 18) for **BM_BufferPool** (p. 5), **BM_MetaPage** (p. 7) type of data is stored in this field

Definition at line 20 of file list.h.

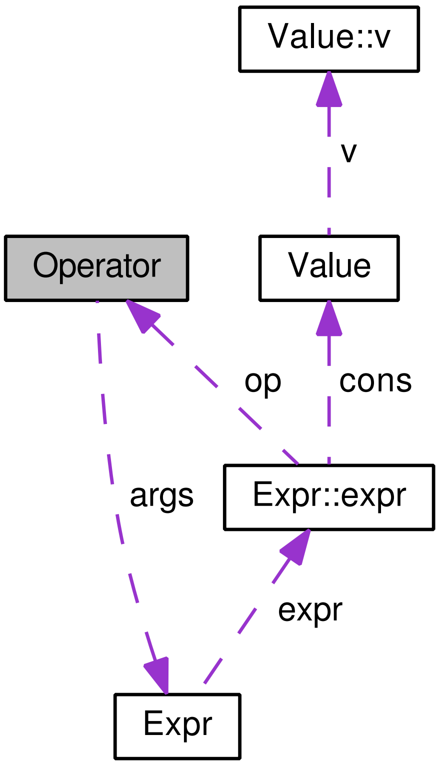
The documentation for this struct was generated from the following file:

- **list.h**

3.14 Operator Struct Reference

```
#include <expr.h>
```

Collaboration diagram for Operator:



Public Attributes

- **OpType type**
- **Expr** ∗∗ **args**

3.14.1 Detailed Description

Definition at line 32 of file expr.h.

3.14.2 Member Data Documentation

3.14.2.1 Expr∗∗ Operator::args

Definition at line 34 of file expr.h.

3.14.2.2 OpType Operator::type

Definition at line 33 of file expr.h.

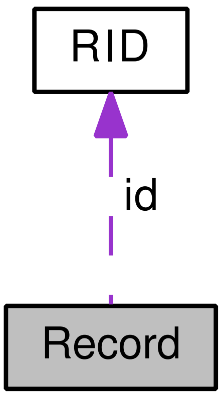
The documentation for this struct was generated from the following file:

- **expr.h**

3.15 Record Struct Reference

```
#include <tables.h>
```

Collaboration diagram for Record:



Public Attributes

- **RID id**
- char ∗ **data**

3.15.1 Detailed Description

A record is any one row of our table.

It has a **RID** (p. 22) id identifier and corresponding data

Definition at line 51 of file tables.h.

3.15.2 Member Data Documentation

3.15.2.1 char∗ Record::data

This contains serialized data of any one row of table.

Definition at line 57 of file tables.h.

3.15.2.2 RID Record::id

This variable stores the relative position of a record within the table.

Definition at line 55 of file tables.h.

The documentation for this struct was generated from the following file:

- **tables.h**

3.16 RID Struct Reference

```
#include <tables.h>
```

Public Attributes

- int **page**
- int **slot**

3.16.1 Detailed Description

Serves as unique id of each record written to the table, and stores the relative position of a record within the table.

Definition at line 39 of file tables.h.

3.16.2 Member Data Documentation

3.16.2.1 int RID::page

This variable contains the page number on which a record is stored.

Definition at line 42 of file tables.h.

3.16.2.2 int RID::slot

Within a page, this variable contains the relative position of the record.

Definition at line 44 of file tables.h.

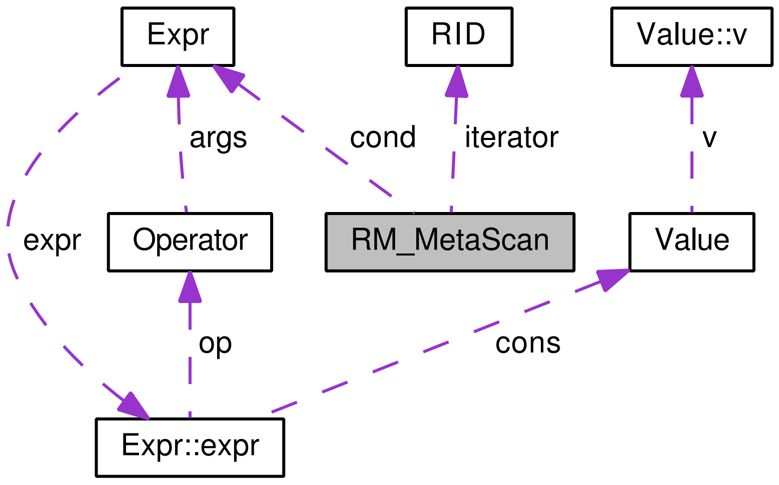
The documentation for this struct was generated from the following file:

- **tables.h**

3.17 RM_MetaScan Struct Reference

```
#include <record_mgr.h>
```

Collaboration diagram for RM_MetaScan:



Public Attributes

- **RID iterator**
- **Expr** ∗ **cond**

3.17.1 Detailed Description

This structure stores the meta data of a scan handle.

Definition at line 26 of file record_mgr.h.

3.17.2 Member Data Documentation

3.17.2.1 Expr∗ RM_MetaScan::cond

This stores the condition that each record must satisfy to qualify in the scan.

Definition at line 32 of file record_mgr.h.

3.17.2.2 RID RM_MetaScan::iterator

This stores the current position of the **RM_ScanHandle** (p. 25) in the table.

Definition at line 30 of file record_mgr.h.

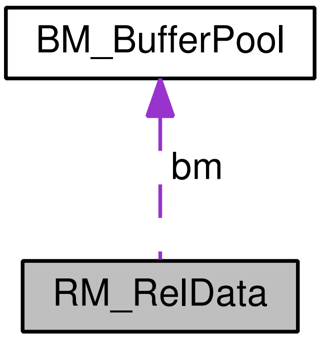
The documentation for this struct was generated from the following file:

- **record_mgr.h**

3.18 RM_RelData Struct Reference

```
#include <tables.h>
```

Collaboration diagram for RM_RelData:



Public Attributes

- int **totalNumberOfRecords**
- int **numDataPages**
- int **firstFreePage**
- **BM_BufferPool** ∗ **bm**

3.18.1 Detailed Description

RelData would be responsible for tracking and managing data and free space of table data.

I have assumed that for each table, the first page contains schema and some meta info. Second page onwards we have page data. In page data I store 2 things before records. 1) a boolean array of size numRecordsPerPage indicating whether the corresponding slot is empty or full. 2) the page number of next free page

Definition at line 100 of file tables.h.

3.18.2 Member Data Documentation

3.18.2.1 BM_BufferPool∗ RM_RelData::bm

This contains a buffer manager for reading and writing pages of the table from disk.

Definition at line 110 of file tables.h.

3.18.2.2 int RM_RelData::firstFreePage

This contains the first page number where not all slots are full.

Definition at line 108 of file tables.h.

3.18.2.3 int RM_RelData::numDataPages

This variable contains the number of data pages that are present in the table.

Definition at line 106 of file tables.h.

3.18.2.4 int RM_RelData::totalNumberOfRecords

This variable contains the total number of records in the table.

Definition at line 104 of file tables.h.

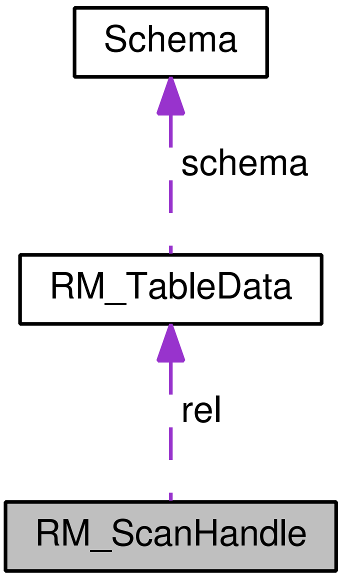
The documentation for this struct was generated from the following file:

- **tables.h**

3.19 RM_ScanHandle Struct Reference

```
#include <record_mgr.h>
```

Collaboration diagram for RM_ScanHandle:



Public Attributes

- **RM_TableData** ∗ **rel**
- void ∗ **mgmtData**

3.19.1 Detailed Description

This structure does the book keeping for scan operations.

The mgmtData stores an **RM_MetaScan** (p. 22), that is mainly responsible for the meta data.

Definition at line 13 of file record_mgr.h.

3.19.2 Member Data Documentation

3.19.2.1 void∗ RM_ScanHandle::mgmtData

This stores the meta data for scan using **RM_MetaScan** (p. 22) struct.

Definition at line 19 of file record_mgr.h.

3.19.2.2 RM_TableData∗ RM_ScanHandle::rel

The table that we are scanning using this **RM_ScanHandle** (p. 25).

Definition at line 17 of file record_mgr.h.

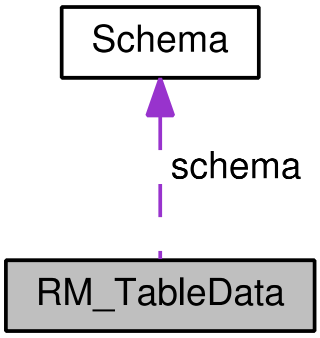
The documentation for this struct was generated from the following file:

- **record_mgr.h**

3.20 RM_TableData Struct Reference

```
#include <tables.h>
```

Collaboration diagram for RM_TableData:



Public Attributes

- char ∗ **name**
- **Schema** ∗ **schema**
- void ∗ **mgmtData**

3.20.1 Detailed Description

TableData: Management Structure for a **Record** (p. 21) Manager to handle one relation.

Definition at line 85 of file tables.h.

3.20.2 Member Data Documentation

3.20.2.1 void∗ RM_TableData::mgmtData

This data is responsible for tracking and managing the table data and free space.

Definition at line 93 of file tables.h.

3.20.2.2 char∗ RM_TableData::name

This variable contains the name of the table.

Definition at line 89 of file tables.h.

3.20.2.3 Schema∗ RM_TableData::schema

This variable contains the schema of the table.

Definition at line 91 of file tables.h.

The documentation for this struct was generated from the following file:

- **tables.h**

3.21 Schema Struct Reference

```
#include <tables.h>
```

Public Attributes

- int **numAttr**
- char ∗∗ **attrNames**
- **DataType** ∗ **dataTypes**
- int ∗ **typeLength**
- int ∗ **keyAttrs**
- int **keySize**

3.21.1 Detailed Description

Schema (p. 27) contains information of a table schema: its attributes, datatypes, sizes of each datatype, key attributes and number of keys.

Definition at line 64 of file tables.h.

3.21.2 Member Data Documentation

3.21.2.1 char∗∗ Schema::attrNames

This variable contains an array of names of each column.

Definition at line 70 of file tables.h.

3.21.2.2 DataType∗ Schema::dataTypes

This variable contains an array of types of each column.

Definition at line 72 of file tables.h.

3.21.2.3 int∗ Schema::keyAttrs

This variable cotains an array of columns to be considered as keys for our table.

Definition at line 76 of file tables.h.

3.21.2.4 int Schema::keySize

This variable contains the number of keys of the table.

Definition at line 78 of file tables.h.

3.21.2.5 int Schema::numAttr

This variable contains the number of columns in our table.

Definition at line 68 of file tables.h.

3.21.2.6 int* Schema::typeLength

This variable contains an array of size of each column.

Definition at line 74 of file tables.h.

The documentation for this struct was generated from the following file:

- **tables.h**

3.22 SM_FileHandle Struct Reference

```
#include <storage_mgr.h>
```

Public Attributes

- char * **fileName**
- int **totalNumPages**
- int **curPagePos**
- void * **mgmtInfo**

3.22.1 Detailed Description

Definition at line 9 of file storage_mgr.h.

3.22.2 Member Data Documentation

3.22.2.1 int SM_FileHandle::curPagePos

Definition at line 12 of file storage_mgr.h.

3.22.2.2 char* SM_FileHandle::fileName

Definition at line 10 of file storage_mgr.h.

3.22.2.3 void* SM_FileHandle::mgmtInfo

Definition at line 13 of file storage_mgr.h.

3.22.2.4 int SM_FileHandle::totalNumPages

Definition at line 11 of file storage_mgr.h.

The documentation for this struct was generated from the following file:

- **storage_mgr.h**

3.23 Value::v Union Reference

`#include <tables.h>`

Public Attributes

- int **intV**
- char ∗ **stringV**
- float **floatV**
- **bool boolV**

3.23.1 Detailed Description

This union is supposed to contain value in one of the 4 union variables which corresponds to the type contained by dt.

Definition at line 27 of file tables.h.

3.23.2 Member Data Documentation

3.23.2.1 bool Value::v::boolV

Definition at line 31 of file tables.h.

3.23.2.2 float Value::v::floatV

Definition at line 30 of file tables.h.

3.23.2.3 int Value::v::intV

Definition at line 28 of file tables.h.

3.23.2.4 char∗ Value::v::stringV

Definition at line 29 of file tables.h.

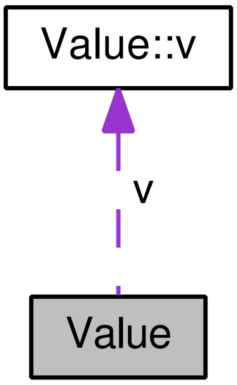
The documentation for this union was generated from the following file:

- **tables.h**

3.24 Value Struct Reference

`#include <tables.h>`

Collaboration diagram for Value:



Classes

- union **v**

Public Attributes

- **DataType dt**
- union **Value::v v**

3.24.1 Detailed Description

Value (p. 29) stores data of any one cell <recordNumber, attrNum>.

The data can be of any one of the types defined in enum DataType

Definition at line 22 of file tables.h.

3.24.2 Member Data Documentation

3.24.2.1 DataType Value::dt

This variable contains the type of the **Value** (p. 29) stored.

Definition at line 25 of file tables.h.

3.24.2.2 union Value::v Value::v

The documentation for this struct was generated from the following file:

- **tables.h**

3.25 VarString Struct Reference

```
#include <tables.h>
```

Public Attributes

- char ∗ **buf**
- int **size**
- int **bufsize**

3.25.1 Detailed Description

dynamic string

Definition at line 118 of file tables.h.

3.25.2 Member Data Documentation

3.25.2.1 char∗ VarString::buf

This variable contains the string variable.

Definition at line 121 of file tables.h.

3.25.2.2 int VarString::bufsize

This variable cotains the size of the buffer.

Definition at line 125 of file tables.h.

3.25.2.3 int VarString::size

This variable contains the size of the string variable.

Definition at line 123 of file tables.h.

The documentation for this struct was generated from the following file:

- **tables.h**

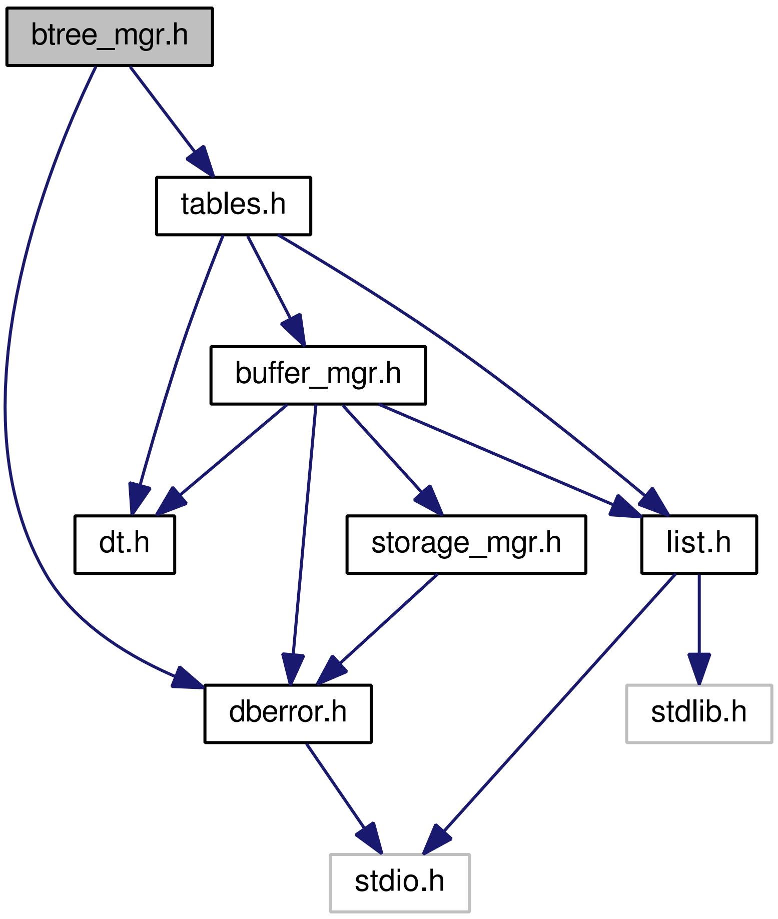
Chapter 4

File Documentation

4.1 btree_mgr.h File Reference

```
#include "dberror.h"
#include "tables.h"
```

Include dependency graph for btree_mgr.h:



Classes

- struct **BTreeHandle**

- struct **BTNode**
- struct **BTreeMeta**
- struct **BT_ScanHandle**
- struct **BT_ScanMeta**

Typedefs

- typedef struct **BTreeHandle BTreeHandle**
- typedef struct **BTNode BTNode**
- typedef struct **BTreeMeta BTreeMeta**
- typedef struct **BT_ScanHandle BT_ScanHandle**
- typedef struct **BT_ScanMeta BT_ScanMeta**

Functions

- **RC initIndexManager** (void ∗mgmtData)
- **RC shutdownIndexManager** ()
- **RC createBtree** (char ∗idxId, **DataType** keyType, int n)
- **RC openBtree** (**BTreeHandle** ∗∗tree, char ∗idxId)
- **RC closeBtree** (**BTreeHandle** ∗tree)
- **RC deleteBtree** (char ∗idxId)
- **RC getNumNodes** (**BTreeHandle** ∗tree, int ∗result)
- **RC getNumEntries** (**BTreeHandle** ∗tree, int ∗result)
- **RC getKeyType** (**BTreeHandle** ∗tree, **DataType** ∗result)
- int **getRootPageNum** (**BTreeHandle** ∗tree)
- int **getFanOut** (**BTreeHandle** ∗tree)
- **RC findKey** (**BTreeHandle** ∗tree, **Value** ∗key, **RID** ∗result)
- **RC recursiveFindKey** (**BTreeHandle** ∗tree, **Value** ∗key, **RID** ∗result, int nodePage)
- **RC insertKey** (**BTreeHandle** ∗tree, **Value** ∗key, **RID** rid)
- **RC recursiveInsertKey** (**BTreeHandle** ∗tree, **Value** ∗key, **RID** rid, int nodePage)
- **RC insertBackPropagate** (**BTreeHandle** ∗tree, **Value** ∗key, int newPage, int parentPage)
- **RC deleteKey** (**BTreeHandle** ∗tree, **Value** ∗key)
- **RC recursiveDeleteKey** (**BTreeHandle** ∗tree, **Value** ∗key, int nodePage)
- **RC deleteBackPropagate** (**BTreeHandle** ∗tree, int nodePage)
- **RC openTreeScan** (**BTreeHandle** ∗tree, **BT_ScanHandle** ∗∗handle)
- **RC nextEntry** (**BT_ScanHandle** ∗handle, **RID** ∗result)
- **RC closeTreeScan** (**BT_ScanHandle** ∗handle)
- char ∗ **printTree** (**BTreeHandle** ∗tree)
- char ∗ **printNode** (**BTreeHandle** ∗tree, int pageNum, int ∗position)
- **RC stringToIdxInfo** (char ∗idxStr, int ∗nEntries, int ∗nNodes, int ∗fOut, **DataType** ∗kType, int ∗rootPage, int ∗maxPage)
- char ∗ **serializeIdxInfo** (**BTreeHandle** ∗tree)
- char ∗ **serializeNode** (**BTNode** ∗node, int n)
- **BTNode** ∗ **stringToNode** (char ∗nodeStr)
- **BTNode** ∗ **readNode** (int nodePage, **BTreeHandle** ∗tree)
- void **writeNode** (**BTNode** ∗node, **BTreeHandle** ∗tree)
- void **writeTreeInfo** (**BTreeHandle** ∗tree)
- void **createNewNodeInternal** (**BTNode** ∗∗node, int fanOut)
- void **createNewNodeLeaf** (**BTNode** ∗∗node, int fanOut)
- void **freeNode** (**BTNode** ∗∗node, **BTreeHandle** ∗tree)
- void **copyValue** (**Value** ∗fromValue, **Value** ∗toValue)

4.1.1 Typedef Documentation

4.1.1.1 typedef struct BT_ScanHandle BT_ScanHandle

This structure stores the details of a tree scan operation.

4.1.1.2 typedef struct BT_ScanMeta BT_ScanMeta

This structure stores an intermediate stage of the scan operation.

It is initialized to represent the first element in BTree (sorted ascending by key).

4.1.1.3 typedef struct BTNode BTNode

This structure represents an individual node of BTree.

4.1.1.4 typedef struct BTreeHandle BTreeHandle

This structure stores a btree with name idxId.

The btree related data is stored in mgmtData.

4.1.1.5 typedef struct BTreeMeta BTreeMeta

This structure stores the details of a BTree.

It is stored in BTreeHandle->mgmtData

4.1.2 Function Documentation

4.1.2.1 RC closeBtree (BTreeHandle $*$ *tree*)

This function Closes a BTree tree, by deallocating and NULL-resetting of the components of the handle.

Parameters

in	*tree*	The tree that needs to be closed. (The buffer manager is deallocated as well).

4.1.2.2 RC closeTreeScan (BT_ScanHandle $*$ *handle*)

This function deallocates and (NULL reset) the elements of a scan handle.

This is to be called at the end of a scan operation.

Parameters

in	*handle*	The scan handle that is to be closed.

4.1.2.3 void copyValue (Value $*$ *fromValue,* Value $*$ *toValue*)

This function copies one **Value** (p. 29) struct onto another.

Parameters

`in`	*fromValue*	**Value** (p. 29) to copy.
`out`	*toValue*	Copied **Value** (p. 29) (copied from fromValue).

4.1.2.4 RC createBtree (char ∗ *idxId,* DataType *keyType,* int *n*)

This function creates an empty BTree of name idxId which has an empty node as root node.

Parameters

`in`	*idxId*	Name of the BTree index file.
`in`	*DataType*	The data type of the key of the btree index.
`in`	*n*	The fanOut of the tree to be created.

4.1.2.5 void createNewNodeInternal (BTNode ∗∗ *node,* int *fanOut*)

This function .

This function allocates memory and returns a pointer to a node structure. It allocates memory to node->childPage and node->keys

Parameters

`in`	*node*	Pointer to pointer to node to be allocated.
`in`	*fanOut*	the fanOut of tree for which this node is being created .

4.1.2.6 void createNewNodeLeaf (BTNode ∗∗ *node,* int *fanOut*)

This function allocates memory and returns a pointer to a node structure.

It allocates memory to node->childRid and node->keys

Parameters

`in`	*node*	Pointer to pointer to node to be allocated.
`in`	*fanOut*	the fanOut of tree for which this node is being created .

4.1.2.7 RC deleteBackPropagate (BTreeHandle ∗ *tree,* int *nodePage*)

This function does the back propagation of underflow during a delete operation.

Parameters

`in`	*tree*	The tree in which the deletion operation is being done.
`in`	*nodePage*	The page number of node from which we start checking for underflow.

4.1.2.8 RC deleteBtree (char ∗ *idxId*)

This function deletes a BTree index file.

Parameters

in	*idxId*	This is the name of the BTree to be deleted.

4.1.2.9 RC deleteKey (BTreeHandle ∗ *tree,* Value ∗ *key*)

This function is used to delete a key in a BTree.

Internally it calls recursiveDeleteKey with root node.

Parameters

in	*tree*	The tree from which the key is to be deleted.
in	*key*	The key that is to be deleted from the tree.

4.1.2.10 RC findKey (BTreeHandle ∗ *tree,* Value ∗ *key,* RID ∗ *result*)

This function is used to find a key in a tree and return its corresponding rid in result.

Internally it calls recursiveFindKey with root node.

Parameters

in	*tree*	The tree in which we are trying to find.
in	*key*	The key which we are trying to find.
out	*result*	Returns the rid corresponding to key, if key is found in the tree.

4.1.2.11 void freeNode (BTNode ∗∗ *node,* BTreeHandle ∗ *tree*)

This function cleans up all the associated memory allocations to a node .

Parameters

in	*node*	Pointer to Pointer to node to be freed.
in	*tree*	Thre tree to which the node belongs. (Needed for fanOut).

4.1.2.12 int getFanOut (BTreeHandle ∗ *tree*)

This function returns the fanOut of the tree.

Parameters

in	*tree*	The tree whose fanOut is being queried.

4.1.2.13 RC getKeyType (BTreeHandle ∗ *tree,* DataType ∗ *result*)

This function returns the key type of the BTree index key .

Parameters

in	*tree*	The tree whose index key type is being queried.

4.1.2.14 RC getNumEntries (BTreeHandle ∗ *tree,* int ∗ *result*)

This function returns the number of rids stored in a BTree.

Parameters

`in`	*tree*	The tree whose numEntries is being queriesd.
`out`	*result*	Used to return the number of entries contained in the btree.

4.1.2.15 RC getNumNodes (BTreeHandle ∗ *tree*, int ∗ *result*)

This function returns the number of nodes stored in a BTree.

Parameters

`in`	*tree*	The tree whose number of nodes is being queried.
`out`	*result*	Used to return the number of nodes contained in the btree.

4.1.2.16 int getRootPageNum (BTreeHandle ∗ *tree*)

This function returns the page number of the root node of tree.

Parameters

`in`	*tree*	The tree whose root page number is being queried.

4.1.2.17 RC initIndexManager (void ∗ *mgmtData*)

This function initializes the index manager.

Parameters

`in`	*mgmtData*	Contains nothing in my implementation.

4.1.2.18 RC insertBackPropagate (BTreeHandle ∗ *tree*, Value ∗ *key*, int *newPage*, int *parentPage*)

This function back propagates any splits that happen on the leaf node, back up to the root node.

Parameters

`in`	*tree*	The tree in which the insert operation is to be back propagated.
`in`	*key*	The key to be inserted in the node (as per back propagation need).
`in`	*newPage*	The page corresponding to key that is to be entered.
`in`	*parentPage*	The pageNumber of node in which [key,newPage] is to be inserted.

4.1.2.19 RC insertKey (BTreeHandle ∗ *tree*, Value ∗ *key*, RID *rid*)

This function is used to insert a key, rid pair into a btree.

Internally it calls recursiveInsertKey starting from root node.

Parameters

`in`	*tree*	The tree in which we want to insert.
`in`	*key*	The key to be inserted.
`in`	*rid*	The rid to be inserted.

4.1.2.20 RC nextEntry (BT_ScanHandle ∗ *handle,* RID ∗ *result*)

This function iterates on each entry of the btree in a sorted order.

Parameters

in	*handle*	The scan handle which contains the current position within the tree.
out	*result*	The next recordId in tree.

4.1.2.21 RC openBtree (BTreeHandle ∗∗ *tree*, char ∗ *idxId*)

This function reads idxId file and stores the corresponding BTree in tree.

Parameters

out	*tree*	Pointer to pointer to tree being read from file.
in	*idxId*	Name of the file on disk, which contains the BTree. (The buffer manager is deallocated as well).

4.1.2.22 RC openTreeScan (BTreeHandle ∗ *tree*, BT_ScanHandle ∗∗ *handle*)

This function initializes a scan handle with details of a btree.

The scan handle acts as an iterator.

Parameters

in	*tree*	The BTree to be scanned.
out	*handle*	The scan handle that is to be initialized at the first element of the tree.

4.1.2.23 char∗ printNode (BTreeHandle ∗ *tree*, int *pageNum*, int ∗ *position*)

This function recursively travels in a pre-order depth first order and prints the subtree starting from node with page number pageNum.

Parameters

in	*tree*	The tree to be printed.
in	*pageNum*	The page number of node whose subtree is to be printed.
in,out	*position*	This variables maintains the pre-order depth first order of each node through the recursive function calls.

4.1.2.24 char∗ printTree (BTreeHandle ∗ *tree*)

This function is used to print a tree.

Internally it uses printNode function to print the subtree of root node.

Parameters

in	*tree*	The tree to be printed.

4.1.2.25 BTNode∗ readNode (int *nodePage*, BTreeHandle ∗ *tree*)

This function is used to read a node from a tree (on disk).

Parameters

in	*node*	The node to be written.
in	*tree*	The tree to which the node is to be written. (Also contains the buffermanager for disk IO)

4.1.2.26 RC recursiveDeleteKey (BTreeHandle ∗ *tree,* Value ∗ *key,* int *nodePage*)

This function recursively goes down to the leaf to delete a given key,rid from the btree.

If needed it calls deleteBackPropagate to correct for underflows starting from leaf up untill the root node.

Parameters

in	*tree*	The tree from which key is to be deleted.
in	*key*	The key to be deleted.
in	*nodePage*	The node from which the backpropagation of underflow is to be started.

4.1.2.27 RC recursiveFindKey (BTreeHandle ∗ *tree,* Value ∗ *key,* RID ∗ *result,* int *nodePage*)

This function recursively finds a key in tree, and returns its value in result.

Parameters

in	*tree*	The tree in which we are trying to find.
in	*key*	The key which we are trying to find.
out	*result*	Returns the rid corresponding to key, if key is found in the tree.
in	*nodePage*	The page number of node from which we are to begin the search.

4.1.2.28 RC recursiveInsertKey (BTreeHandle ∗ *tree,* Value ∗ *key,* RID *rid,* int *nodePage*)

This function inserts a key, rid pair in a BTree subtree starting at node with page number nodePage.

Parameters

in	*tree*	The tree in which we want to insert.
in	*key*	The key that we want to insert.
in	*rid*	The rid that we want to insert in subtree starting at node with page number nodePage.
in	*nodePage*	Page number of the node where the subtree begins. The rid that we want to insert.

4.1.2.29 char∗ serializeIdxInfo (BTreeHandle ∗ *tree*)

This function creates the string equivalent of the tree meta info.

This string is supposed to be the first page of any index file.

Parameters

in	*tree*	The tree for which we are creating the tree info string .

4.1.2.30 char∗ serializeNode (BTNode ∗ *node,* int *n*)

This function converts a node to an equivalent string representation, which can be potentially written to disk.

Parameters

`in`	*node*	The node to be converted to string.
`in`	*n*	The fanOut of the tree to which this node belongs.

4.1.2.31 RC shutdownIndexManager ( )

This function closes the index manager.

Contains nothing in my implementation.

4.1.2.32 RC stringToIdxInfo (char ∗ *idxStr,* int ∗ *nEntries,* int ∗ *nNodes,* int ∗ *fOut,* DataType ∗ *kType,* int ∗ *rootPage,* int ∗ *maxPage*)

This function parses the string of first page of an index file, and returns all the parameters of the index accordingly.

Parameters

`in`	*idxStr*	The string to be parsed.
`out`	*nEntries*	The number of entries in the BTree.
`out`	*nNodes*	The number of node in the BTree.
`out`	*fOut*	The fanOut of the BTree.
`out`	*kTypoe*	The DataType of the key for this BTree.
`out`	*rootPage*	The page number of the root node of the BTree.
`out`	*maxPage*	The maximum page number that any node has in the BTree.

4.1.2.33 BTNode∗ stringToNode (char ∗ *nodeStr*)

This function reads a node page and converts it into a node object.

Parameters

`in`	*nodeStr*	Node string as read from a node page. This node string will be converted to a node object.

4.1.2.34 void writeNode (BTNode ∗ *node,* BTreeHandle ∗ *tree*)

This function writes a node to disk.

For writing to disk, the buffer manager bm (in **BTreeMeta** (p. 14), in **BTreeHandle** (p. 13)) is used.

Parameters

`in`	*node*	The node to be written.
`in`	*tree*	The tree to which the node is to be written. (Also contains the buffermanager for disk IO)

4.1.2.35 void writeTreeInfo (BTreeHandle ∗ *tree*)

This function saves the meta info of a given tree on page 0 of a BTree index page.

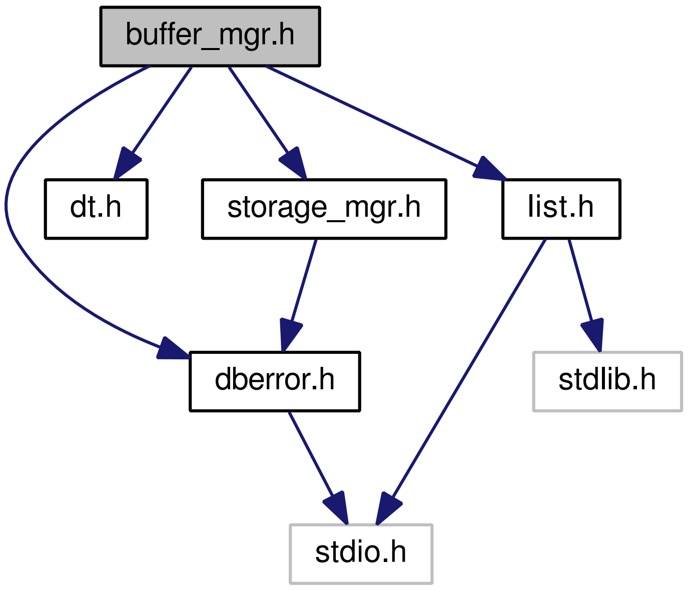
Parameters

in	*tree*	The tree whose meta info is to be saved.

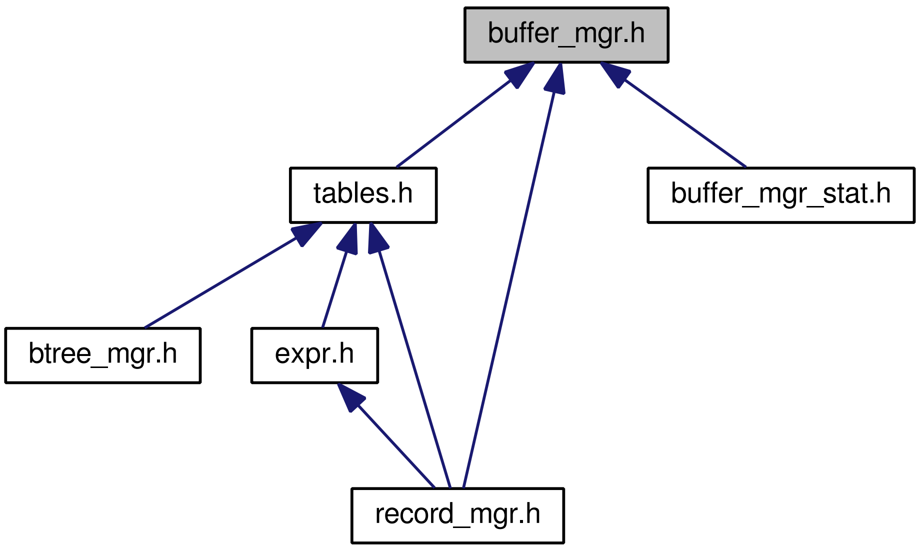
4.2 buffer_mgr.h File Reference

```
#include "dberror.h"
#include "dt.h"
#include "list.h"
#include "storage_mgr.h"
```

Include dependency graph for buffer_mgr.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct **BM_BufferPool**
- struct **BM_PageHandle**
- struct **BM_MetaPage**
- struct **BM_MetaList**

Macros

- #define **NO_PAGE** -1
- #define **MAKE_POOL**() ((**BM_BufferPool** ∗) malloc (sizeof(**BM_BufferPool**)))
- #define **MAKE_PAGE_HANDLE**() ((**BM_PageHandle** ∗) malloc (sizeof(**BM_PageHandle**)))
- #define **MAKE_META_PAGE**() ((**BM_MetaPage** ∗) malloc (sizeof(**BM_MetaPage**)))
- #define **MAKE_META_LIST**() ((**BM_MetaList** ∗) malloc (sizeof(**BM_MetaList**)))

Typedefs

- typedef enum **ReplacementStrategy ReplacementStrategy**
- typedef int **PageNumber**
- typedef struct **BM_BufferPool BM_BufferPool**
- typedef struct **BM_PageHandle BM_PageHandle**
- typedef struct **BM_MetaPage BM_MetaPage**
- typedef struct **BM_MetaList BM_MetaList**

Enumerations

- enum **ReplacementStrategy** {
  RS_FIFO = 0, **RS_LRU** = 1, **RS_CLOCK** = 2, **RS_LFU** = 3,
  RS_LRU_K = 4 }

Functions

- **RC initBufferPool** (**BM_BufferPool** ∗const bm, const char ∗const pageFileName, const int numPages, **ReplacementStrategy** strategy, void ∗stratData)
- **RC shutdownBufferPool** (**BM_BufferPool** ∗const bm)
- **RC forceFlushPool** (**BM_BufferPool** ∗const bm)
- **RC markDirty** (**BM_BufferPool** ∗const bm, **BM_PageHandle** ∗const page)
- **RC unpinPage** (**BM_BufferPool** ∗const bm, **BM_PageHandle** ∗const page)
- **RC forcePage** (**BM_BufferPool** ∗const bm, **BM_PageHandle** ∗const page)
- **RC pinPage** (**BM_BufferPool** ∗const bm, **BM_PageHandle** ∗const page, const **PageNumber** pageNum)
- **RC fifoAddPage** (**BM_BufferPool** ∗const bm, const **PageNumber** pageNum)
- **RC lruAddPage** (**BM_BufferPool** ∗const bm, const **PageNumber** pageNum)
- **PageNumber** ∗ **getFrameContents** (**BM_BufferPool** ∗const bm)
- **bool** ∗ **getDirtyFlags** (**BM_BufferPool** ∗const bm)
- int ∗ **getFixCounts** (**BM_BufferPool** ∗const bm)
- int **getNumReadIO** (**BM_BufferPool** ∗const bm)
- int **getNumWriteIO** (**BM_BufferPool** ∗const bm)

4.2.1 Macro Definition Documentation

4.2.1.1 #define MAKE_META_LIST( ) ((BM_MetaList ∗) malloc (sizeof(BM_MetaList)))

Definition at line 100 of file buffer_mgr.h.

4.2.1.2 #define MAKE_META_PAGE( ) ((BM_MetaPage ∗) malloc (sizeof(BM_MetaPage)))

Definition at line 97 of file buffer_mgr.h.

4.2.1.3 #define MAKE_PAGE_HANDLE( ) ((BM_PageHandle ∗) malloc (sizeof(BM_PageHandle)))

Definition at line 94 of file buffer_mgr.h.

4.2.1.4 #define MAKE_POOL( ) ((BM_BufferPool ∗) malloc (sizeof(BM_BufferPool)))

Definition at line 91 of file buffer_mgr.h.

4.2.1.5 #define NO_PAGE -1

Definition at line 29 of file buffer_mgr.h.

4.2.2 Typedef Documentation

4.2.2.1 typedef struct BM_BufferPool BM_BufferPool

BM_BufferPool (p. 5) creates an in memory pool of pages which are read using storage manager.

The addition and deletion of pages in the pool is done based on the value of "strategy" All the pages are stored in mgmt data using a **BM_MetaList** (p. 6) struct object.

4.2.2.2 typedef struct BM_MetaList BM_MetaList

This structure contains some book keeping information for the buffer pool, along with the pages in the pool.

4.2.2.3 typedef struct BM_MetaPage BM_MetaPage

This structures combines some basic meta information to a pageHandle, that allows us to maintain some book-keeping information for a given pageHandle.

4.2.2.4 typedef struct BM_PageHandle BM_PageHandle

BM_PageHandle (p. 8) budles a PageNumber (int) pageNum field along with its page contents in field "data".

4.2.2.5 typedef int PageNumber

PageNumber is of int type.

Definition at line 28 of file buffer_mgr.h.

4.2.2.6 typedef enum ReplacementStrategy ReplacementStrategy

An enum to capture different replacement strategies.

4.2.3 Enumeration Type Documentation

4.2.3.1 enum ReplacementStrategy

An enum to capture different replacement strategies.

Enumerator

RS_FIFO

RS_LRU

RS_CLOCK

RS_LFU

RS_LRU_K

Definition at line 16 of file buffer_mgr.h.

4.2.4 Function Documentation

4.2.4.1 RC fifoAddPage (BM_BufferPool ∗const *bm,* const PageNumber *pageNum*)

This function is called from within pinPage.

The pool is created using a liked list (**linkedList** (p. 18)). This function performs addition as well as deletion of pages based on fifo logic. If the pool has empty slots, this function only adds pages. If the pool has no empty slots, the first a deletion is performed and then an addition is performed. The additions are done at the tail of the linked list. The deletions are done at the head. This keeps the linked list sorted in fifo order. After deletion of a node all its memory is deallocated.

Parameters

`in`	*bm*	The buffer pool which contains this page
`in`	*pageNum*	The page number that we want to add to pool.

4.2.4.2 RC forceFlushPool (BM_BufferPool ∗const *bm*)

Iterates through all the pages of the bufferpool, checks if the page is dirty and if it is, then it writes the pages to disk using storage_mgr function writeBlock.

Parameters

`in`	*bm*	The buffer pool to force flush on disk

4.2.4.3 RC forcePage (BM_BufferPool ∗const *bm,* BM_PageHandle ∗const *page*)

If the page is dirty, this function forces the page to be written on disk.

Parameters

`in`	*bm*	The buffer pool which contains this page
`in`	*bm*	The handle of the page that we want to force on disk

4.2.4.4 bool∗ getDirtyFlags (BM_BufferPool ∗const *bm*)

Returns an array of type bool of length = size of buffer pool.

Each bool entry of this array is 1 if the corresponding page is dirty, else it is 0.

Parameters

in	*bm*	The buffer pool which we are querying

4.2.4.5 int∗ getFixCounts (BM_BufferPool ∗const *bm*)

Returns an array of length = size of buffer pool.

Each integer entry contains the fixCount of the corresponding page in buffer pool.

Parameters

in	*bm*	The buffer pool which we are querying

4.2.4.6 PageNumber∗ getFrameContents (BM_BufferPool ∗const *bm*)

Returns an array of page numbers currently contained in the buffer pool.

Parameters

in	*bm*	The buffer pool which we are querying

4.2.4.7 int getNumReadIO (BM_BufferPool ∗const *bm*)

Returns the number of times the buffer pool has had Read IO since its initialization.

Parameters

in	*bm*	The buffer pool which we are querying

4.2.4.8 int getNumWriteIO (BM_BufferPool ∗const *bm*)

Returns the number of times the buffer pool has had Write IO since its initialization.

Parameters

in	*bm*	The buffer pool which we are querying

4.2.4.9 RC initBufferPool (BM_BufferPool ∗const *bm,* const char ∗const *pageFileName,* const int *numPages,* ReplacementStrategy *strategy,* void ∗ *stratData*)

Initializes a buffer pool, and allocates memory to pages.

Parameters

in	*bm*	The buffer pool to initialize.
in	*pageFileName*	The name of the pageFile for which we are initializing this instance of buffer manager
in	*numPages*	The number of pages that we want to keep in the pool
in	*strategy*	Specifies which paging strategy to use adding and deleting pages from the pool

in	*stratData*	contains additional information for strategy

4.2.4.10 RC lruAddPage (BM_BufferPool ∗const *bm*, const PageNumber *pageNum*)

This function is called from within pinPage.

The pool is created using a liked list (**linkedList** (p. 18)). This function performs addition as well as deletion of pages based on LRU logic. If the pool has empty slots, this function only adds pages. If the pool has no empty slots, the first a deletion is performed and then an addition is performed. The additions are done at the tail of the linked list. The deletions are done at the head. If the page is already present in the list, even then the page is first removed from the list and then added at the tail. This keeps the linked list sorted in LRU order (with the page at head always being the least recently used). After deletion of a node all its memory is deallocated.

Parameters

in	*bm*	The buffer pool which contains this page
in	*pageNum*	The page number that we want to add to pool.

4.2.4.11 RC markDirty (BM_BufferPool ∗const *bm*, BM_PageHandle ∗const *page*)

Tags a page as dirty, signifying that the current contents of the page might be different from what resides on disk.

Parameters

in	*bm*	The buffer pool which contains this page
in	*page*	The **BM_PageHandle** (p. 8) of the page that we want to mark as dirty.

4.2.4.12 RC pinPage (BM_BufferPool ∗const *bm*, BM_PageHandle ∗const *page*, const PageNumber *pageNum*)

Increments the fixCount of the page by 1.

This signifies that one more process is now accesesing this page in the buffer pool

Parameters

in	*bm*	The buffer pool which contains this page
out	*page*	The handle of the page indicated by pageNum
in	*pageNum*	The page number that we want to pin.

4.2.4.13 RC shutdownBufferPool (BM_BufferPool ∗const *bm*)

Deallocates memory allotted the contents of the buffer pool.

Parameters

in	*bm*	The buffer pool that we want to de allocate.

4.2.4.14 RC unpinPage (BM_BufferPool ∗const *bm*, BM_PageHandle ∗const *page*)

Reduces the fixCount of the page by 1 (if fixCount $>$0).

This signifies that one of the processes is no longer accessing this page

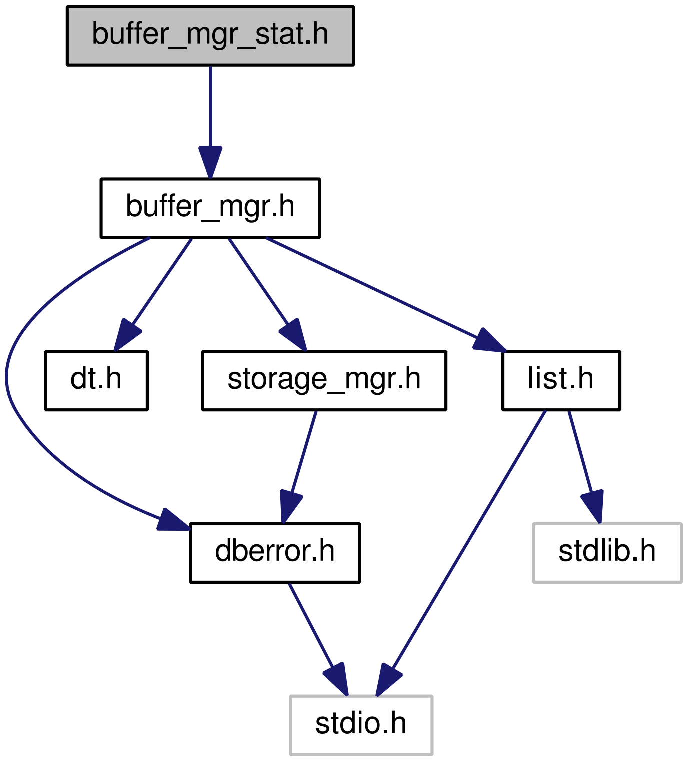
Parameters

`in`	*bm*	The buffer pool which contains this page &
`in`	*page*	The handle of the page that we want to unpin.

4.3 buffer_mgr_stat.h File Reference

```
#include "buffer_mgr.h"
```

Include dependency graph for buffer_mgr_stat.h:



Functions

- void **printPoolContent** (**BM_BufferPool** ∗const bm)
- void **printPageContent** (**BM_PageHandle** ∗const page)
- char ∗ **sprintPoolContent** (**BM_BufferPool** ∗const bm)
- char ∗ **sprintPageContent** (**BM_PageHandle** ∗const page)

4.3.1 Function Documentation

4.3.1.1 void printPageContent (**BM_PageHandle** ∗const *page*)

4.3.1.2 void printPoolContent (**BM_BufferPool** ∗const *bm*)

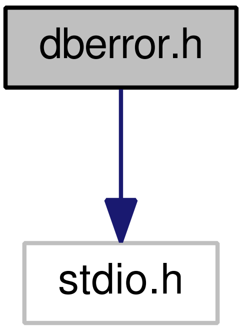
4.3.1.3 char∗ sprintPageContent (**BM_PageHandle** ∗const *page*)

4.3.1.4 char∗ sprintPoolContent (**BM_BufferPool** ∗const *bm*)

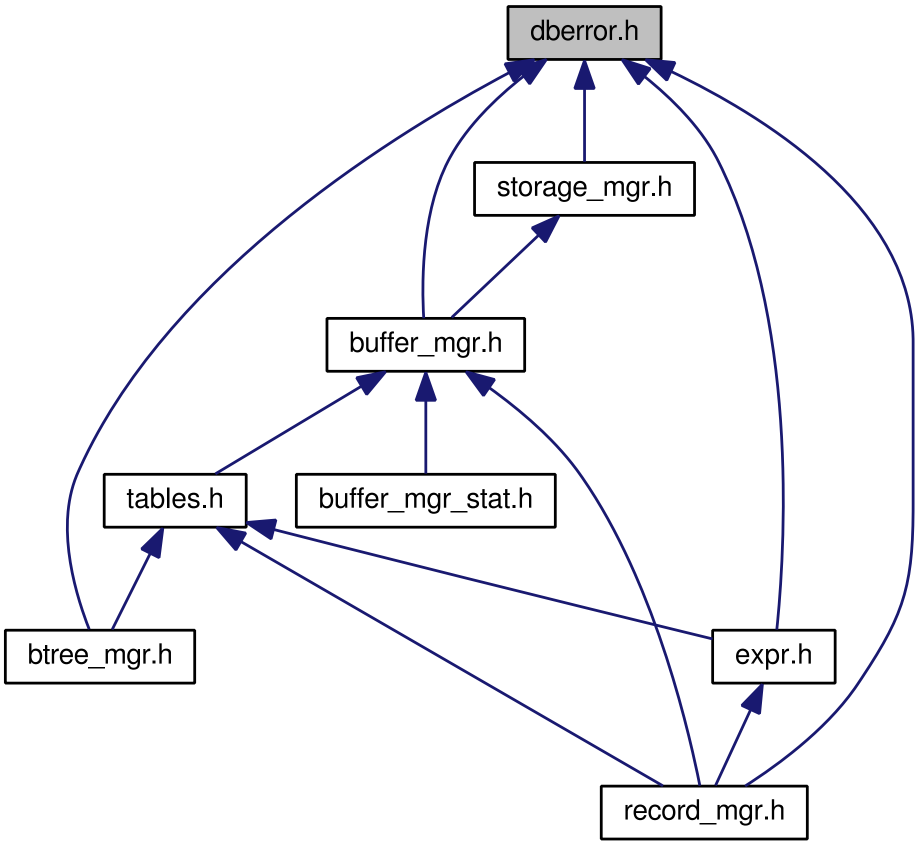
4.4 dberror.h File Reference

`#include "stdio.h"`

Include dependency graph for dberror.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define **META_SIZE** 512
- #define **PAGE_SIZE** 4096
- #define **TABLE_META_SIZE** 512
- #define **RM_BUFFER_SIZE** 5
- #define **IDX_BUFFER_SIZE** 3

- #define **SERIALIZED_INT** 5
- #define **minINF** -10000
- #define **RC_OK** 0
- #define **RC_FILE_NOT_FOUND** 1
- #define **RC_FILE_HANDLE_NOT_INIT** 2
- #define **RC_WRITE_FAILED** 3
- #define **RC_READ_NON_EXISTING_PAGE** 4
- #define **RC_FILE_ALREADY_EXISTS** 5
- #define **RC_BM_POOL_HAS_PINNED_PAGES** 101
- #define **RC_BM_ALL_PAGES_PINNED** 102
- #define **RC_RM_COMPARE_VALUE_OF_DIFFERENT_DATATYPE** 200
- #define **RC_RM_EXPR_RESULT_IS_NOT_BOOLEAN** 201
- #define **RC_RM_BOOLEAN_EXPR_ARG_IS_NOT_BOOLEAN** 202
- #define **RC_RM_NO_MORE_TUPLES** 203
- #define **RC_RM_NO_PRINT_FOR_DATATYPE** 204
- #define **RC_RM_UNKOWN_DATATYPE** 205
- #define **RC_RM_ASSIGN_VALUE_OF_DIFFERENT_DATATYPE** 206
- #define **RC_RM_CANNOT_CLOSE_TABLE_IN_USE** 207
- #define **RC_RM_RECORD_NOT_FOUND** 208
- #define **RC_IM_KEY_NOT_FOUND** 300
- #define **RC_IM_KEY_ALREADY_EXISTS** 301
- #define **RC_IM_N_TO_LAGE** 302
- #define **RC_IM_NO_MORE_ENTRIES** 303
- #define **RC_IM_CANNOT_CLOSE_IDX_IN_USE** 304
- #define **THROW**(rc, message)
- #define **CHECK**(code)

Typedefs

- typedef int **RC**

Functions

- void **printError** (**RC** error)
- char ∗ **errorMessage** (**RC** error)

Variables

- char ∗ **RC_message**

4.4.1 Macro Definition Documentation

4.4.1.1 #define CHECK(*code*)

Value:

```
do {                                   \
    int rc_internal = (code);                      \
    if (rc_internal != RC_OK)                      \
      {                                \
    char *message = errorMessage(rc_internal);          \
    printf("[%s-L%i-%s] ERROR: Operation returned error: %s\n",__FILE__, __LINE__, __TIME__, message); \
    free(message);                          \
    exit(1);                           \
      }                                \
  } while(0);
```

Definition at line 58 of file dberror.h.

4.4.1.2 #define IDX_BUFFER_SIZE 3

Definition at line 11 of file dberror.h.

4.4.1.3 #define META_SIZE 512

Definition at line 7 of file dberror.h.

4.4.1.4 #define minINF -10000

Definition at line 13 of file dberror.h.

4.4.1.5 #define PAGE_SIZE 4096

Definition at line 8 of file dberror.h.

4.4.1.6 #define RC_BM_ALL_PAGES_PINNED 102

Definition at line 26 of file dberror.h.

4.4.1.7 #define RC_BM_POOL_HAS_PINNED_PAGES 101

Definition at line 25 of file dberror.h.

4.4.1.8 #define RC_FILE_ALREADY_EXISTS 5

Definition at line 23 of file dberror.h.

4.4.1.9 #define RC_FILE_HANDLE_NOT_INIT 2

Definition at line 20 of file dberror.h.

4.4.1.10 #define RC_FILE_NOT_FOUND 1

Definition at line 19 of file dberror.h.

4.4.1.11 #define RC_IM_CANNOT_CLOSE_IDX_IN_USE 304

Definition at line 42 of file dberror.h.

4.4.1.12 #define RC_IM_KEY_ALREADY_EXISTS 301

Definition at line 39 of file dberror.h.

4.4.1.13 #define RC_IM_KEY_NOT_FOUND 300

Definition at line 38 of file dberror.h.

4.4.1.14 #define RC_IM_N_TO_LAGE 302

Definition at line 40 of file dberror.h.

4.4.1.15 #define RC_IM_NO_MORE_ENTRIES 303

Definition at line 41 of file dberror.h.

4.4.1.16 #define RC_OK 0

Definition at line 18 of file dberror.h.

4.4.1.17 #define RC_READ_NON_EXISTING_PAGE 4

Definition at line 22 of file dberror.h.

4.4.1.18 #define RC_RM_ASSIGN_VALUE_OF_DIFFERENT_DATATYPE 206

Definition at line 34 of file dberror.h.

4.4.1.19 #define RC_RM_BOOLEAN_EXPR_ARG_IS_NOT_BOOLEAN 202

Definition at line 30 of file dberror.h.

4.4.1.20 #define RC_RM_CANNOT_CLOSE_TABLE_IN_USE 207

Definition at line 35 of file dberror.h.

4.4.1.21 #define RC_RM_COMPARE_VALUE_OF_DIFFERENT_DATATYPE 200

Definition at line 28 of file dberror.h.

4.4.1.22 #define RC_RM_EXPR_RESULT_IS_NOT_BOOLEAN 201

Definition at line 29 of file dberror.h.

4.4.1.23 #define RC_RM_NO_MORE_TUPLES 203

Definition at line 31 of file dberror.h.

4.4.1.24 #define RC_RM_NO_PRINT_FOR_DATATYPE 204

Definition at line 32 of file dberror.h.

4.4.1.25 #define RC_RM_RECORD_NOT_FOUND 208

Definition at line 36 of file dberror.h.

4.4.1.26 #define RC_RM_UNKOWN_DATATYPE 205

Definition at line 33 of file dberror.h.

4.4.1.27 #define RC_WRITE_FAILED 3

Definition at line 21 of file dberror.h.

4.4.1.28 #define RM_BUFFER_SIZE 5

Definition at line 10 of file dberror.h.

4.4.1.29 #define SERIALIZED_INT 5

Definition at line 12 of file dberror.h.

4.4.1.30 #define TABLE_META_SIZE 512

Definition at line 9 of file dberror.h.

4.4.1.31 #define THROW(*rc,* *message*)

Value:

```
do {                   RC_message=message;   \
    return rc;         \
  } while (0)          \
```

Definition at line 51 of file dberror.h.

4.4.2 Typedef Documentation

4.4.2.1 typedef int RC

Definition at line 16 of file dberror.h.

4.4.3 Function Documentation

4.4.3.1 char∗ errorMessage (RC *error*)

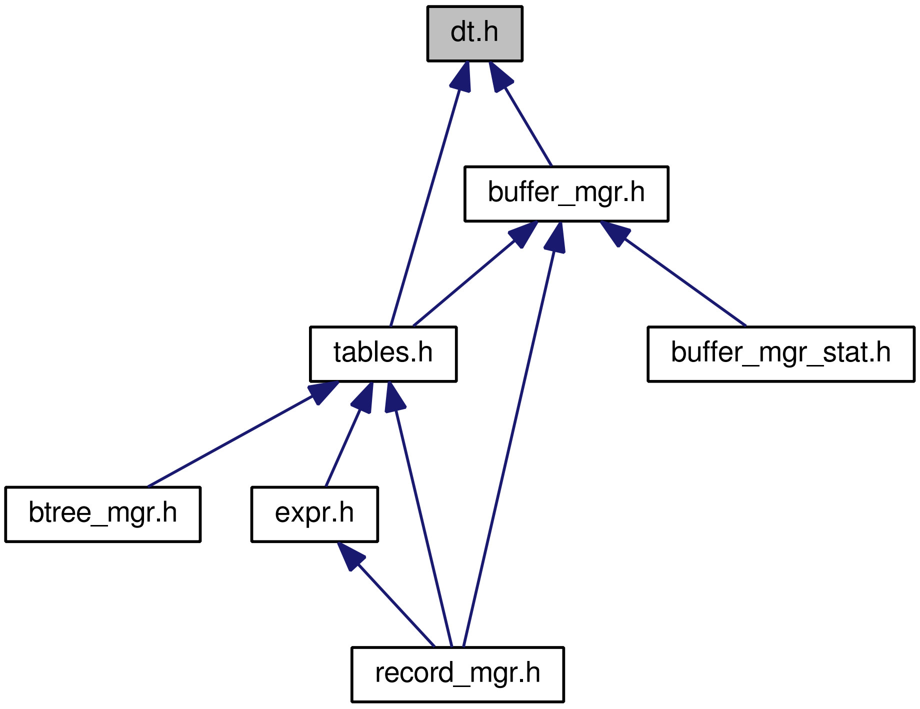
4.4.3.2 void printError (RC *error*)

4.4.4 Variable Documentation

4.4.4.1 char∗ RC_message

4.5 dt.h File Reference

This graph shows which files directly or indirectly include this file:



Macros

- #define **true** 1
- #define **false** 0
- #define **TRUE true**
- #define **FALSE false**

Typedefs

- typedef short **bool**

4.5.1 Macro Definition Documentation

4.5.1.1 #define false 0

Definition at line 8 of file dt.h.

4.5.1.2 #define FALSE false

Definition at line 12 of file dt.h.

4.5.1.3 #define true 1

Definition at line 7 of file dt.h.

4.5.1.4 #define TRUE true

Definition at line 11 of file dt.h.

4.5.2 Typedef Documentation

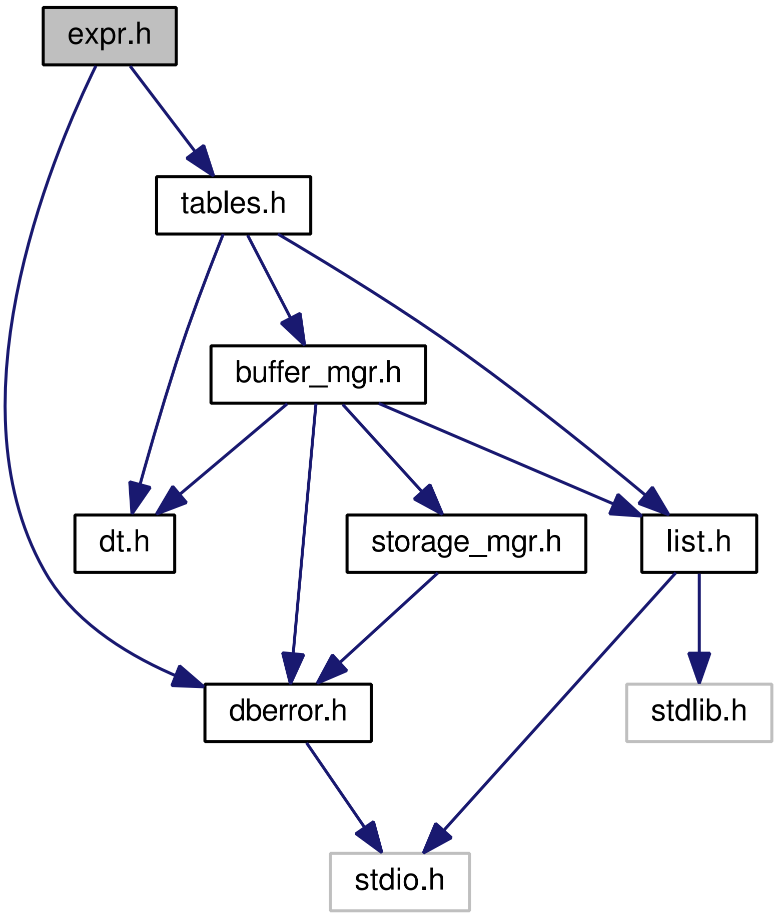
4.5.2.1 typedef short bool

Definition at line 6 of file dt.h.

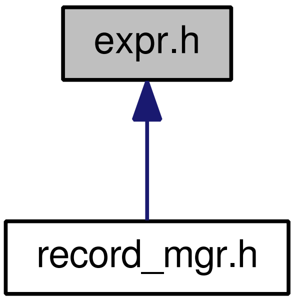
4.6 expr.h File Reference

```
#include "dberror.h"
#include "tables.h"
```

Include dependency graph for expr.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct **Expr**
- union **Expr::expr**
- struct **Operator**

Macros

- #define **CPVAL**(_result, _input)
- #define **MAKE_BINOP_EXPR**(_result, _left, _right, _optype)
- #define **MAKE_UNOP_EXPR**(_result, _input, _optype)
- #define **MAKE_ATTRREF**(_result, _attr)
- #define **MAKE_CONS**(_result, _value)

Typedefs

- typedef enum **ExprType ExprType**
- typedef struct **Expr Expr**
- typedef enum **OpType OpType**
- typedef struct **Operator Operator**

Enumerations

- enum **ExprType** { **EXPR_OP**, **EXPR_CONST**, **EXPR_ATTRREF** }
- enum **OpType** {
  OP_BOOL_AND, **OP_BOOL_OR**, **OP_BOOL_NOT**, **OP_COMP_EQUAL**,
  OP_COMP_SMALLER }

Functions

- **RC valueEquals** (**Value** ∗left, **Value** ∗right, **Value** ∗result)
- **RC valueSmaller** (**Value** ∗left, **Value** ∗right, **Value** ∗result)
- **RC boolNot** (**Value** ∗input, **Value** ∗result)
- **RC boolAnd** (**Value** ∗left, **Value** ∗right, **Value** ∗result)
- **RC boolOr** (**Value** ∗left, **Value** ∗right, **Value** ∗result)
- **RC evalExpr** (**Record** ∗record, **Schema** ∗schema, **Expr** ∗expr, **Value** ∗∗result)
- **RC freeExpr** (**Expr** ∗expr)
- void **freeVal** (**Value** ∗val)

4.6.1 Macro Definition Documentation

4.6.1.1 #define CPVAL(*_result*, *_input*)

Value:

```
do {                                    \
    (_result)->dt = _input->dt;                       \
  switch(_input->dt)                        \
    {                                   \
    case DT_INT:                            \
      (_result)->v.intV = _input->v.intV;                   \
      break;                            \
    case DT_STRING:                         \
      (_result)->v.stringV = (char *) malloc(strlen(_input->v.stringV));    \
      strcpy((_result)->v.stringV, _input->v.stringV);          \
      break;                            \
    case DT_FLOAT:                          \
      (_result)->v.floatV = _input->v.floatV;               \
      break;                            \
    case DT_BOOL:                           \
      (_result)->v.boolV = _input->v.boolV;             \
      break;                            \
    }                                   \
} while(0)
```

Definition at line 48 of file expr.h.

4.6.1.2 #define MAKE_ATTRREF(*_result*, *_attr*)

Value:

```
do {                                    \
    _result = (Expr *) malloc(sizeof(Expr));                \
    _result->type = EXPR_ATTRREF;                     \
    _result->expr.attrRef = _attr;                    \
  } while(0)
```

Definition at line 92 of file expr.h.

4.6.1.3 #define MAKE_BINOP_EXPR(*_result*, *_left*, *_right*, *_optype*)

Value:

```
do {                                       Operator *_op = (Operator *) malloc(sizeof(Operator));        \
      _result = (Expr *) malloc(sizeof(Expr));              \
      _result->type = EXPR_OP;                      \
      _result->expr.op = _op;                       \
      _op->type = _optype;                      \
      _op->args = (Expr **) malloc(2 * sizeof(Expr*));          \
      _op->args[0] = _left;                     \
      _op->args[1] = _right;                        \
    } while (0)
```

Definition at line 69 of file expr.h.

4.6.1.4 #define MAKE_CONS(*_result*, *_value*)

Value:

```
do {                                    \
    _result = (Expr *) malloc(sizeof(Expr));                \
    _result->type = EXPR_CONST;                       \
    _result->expr.cons = _value;                      \
  } while(0)
```

Definition at line 99 of file expr.h.

4.6.1.5 #define MAKE_UNOP_EXPR(*_result*, *_input*, *_optype*)

Value:

```
do {                                          Operator *_op = (Operator *) malloc(sizeof(Operator));      \
    _result = (Expr *) malloc(sizeof(Expr));                    \
    _result->type = EXPR_OP;                            \
    _result->expr.op = _op;                         \
    _op->type = _optype;                            \
    _op->args = (Expr **) malloc(sizeof(Expr*));                \
    _op->args[0] = _input;                          \
  } while (0)
```

Definition at line 81 of file expr.h.

4.6.2 Typedef Documentation

4.6.2.1 typedef struct Expr Expr

4.6.2.2 typedef enum ExprType ExprType

4.6.2.3 typedef struct Operator Operator

4.6.2.4 typedef enum OpType OpType

4.6.3 Enumeration Type Documentation

4.6.3.1 enum ExprType

Enumerator

EXPR_OP

EXPR_CONST

EXPR_ATTRREF

Definition at line 8 of file expr.h.

4.6.3.2 enum OpType

Enumerator

OP_BOOL_AND

OP_BOOL_OR

OP_BOOL_NOT

OP_COMP_EQUAL

OP_COMP_SMALLER

Definition at line 24 of file expr.h.

4.6.4 Function Documentation

4.6.4.1 RC boolAnd (Value ∗ *left*, Value ∗ *right*, Value ∗ *result*)

4.6.4.2 RC boolNot (Value ∗ *input*, Value ∗ *result*)

4.6.4.3 RC boolOr (Value ∗ *left*, Value ∗ *right*, Value ∗ *result*)

4.6.4.4 RC evalExpr (**Record** ∗ *record,* **Schema** ∗ *schema,* **Expr** ∗ *expr,* **Value** ∗∗ *result*)

4.6.4.5 RC freeExpr (**Expr** ∗ *expr*)

4.6.4.6 void freeVal (**Value** ∗ *val*)

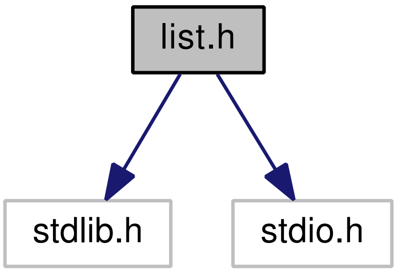
4.6.4.7 RC valueEquals (**Value** ∗ *left,* **Value** ∗ *right,* **Value** ∗ *result*)

4.6.4.8 RC valueSmaller (**Value** ∗ *left,* **Value** ∗ *right,* **Value** ∗ *result*)

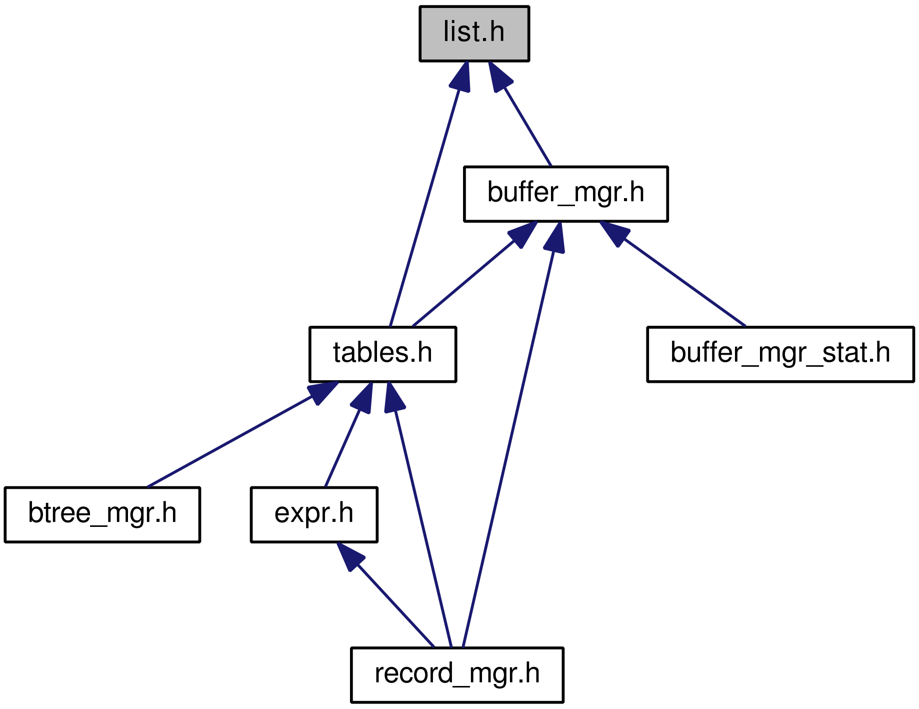
4.7 list.h File Reference

```
#include <stdlib.h>
#include <stdio.h>
```

Include dependency graph for list.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct **linkedListNode**

- struct **linkedList**

Typedefs

- typedef struct **linkedListNode** **linkedListNode**
- typedef struct **linkedList** **linkedList**

Functions

- **linkedList** ∗ **listCreate** ()
- void **listDestroy** (**linkedList** ∗list)
- void **listClear** (**linkedList** ∗list)
- void **listClearDestroy** (**linkedList** ∗list)
- void **listAddToEnd** (**linkedList** ∗list, void ∗value)
- void ∗ **listDequeue** (**linkedList** ∗list)
- void **listPush** (**linkedList** ∗list, void ∗value)
- void ∗ **listPop** (**linkedList** ∗list)
- void ∗ **listRemove** (**linkedList** ∗list, **linkedListNode** ∗node)

4.7.1 Typedef Documentation

4.7.1.1 typedef struct linkedList linkedList

This is a doubly linked list for storing pages in the buffer pool.

Doubly linked list allows me the flexibility to add and remove at both ends. This allows me to do basic operations like fifo additions in constant time (independent of the length of the linked list).

4.7.1.2 typedef struct linkedListNode linkedListNode

This is a node for a doubly linked list **linkedList** (p. 18).

it contains its data as (void∗).

4.7.2 Function Documentation

4.7.2.1 void listAddToEnd (linkedList ∗ *list,* void ∗ *value*)

Adds a new node containing data ∗value at the tail of the list.

Parameters

`in`	*list*	The list to which we want to add
`in`	*value*	The value that we want to put in the new node (added at tail).

4.7.2.2 void listClear (linkedList ∗ *list*)

Iterates on each node of the list and deallocates the data (void ∗value) in the node.

Parameters

in	*list*	The list that we want to clear

4.7.2.3 void listClearDestroy (**linkedList** ∗ *list*)

First it clears the list using listClear, and then it destroys the list using listDestroy.

Parameters

in	*list*	The list that we want to clear and destroy

4.7.2.4 **linkedList**∗ listCreate ( )

Creates and empty list with both head and tail pointing to NULL.

4.7.2.5 void∗ listDequeue (**linkedList** ∗ *list*)

Removes a node from the tail end of the list.

Parameters

in	*list*	The list that we want to dequeue (at tail)

4.7.2.6 void listDestroy (**linkedList** ∗ *list*)

Iterates on each node and deallocates the node and then de allocates the list.

Parameters

in	*list*	The list that we want to deallocate and destroy.

4.7.2.7 void∗ listPop (**linkedList** ∗ *list*)

Removes a node from the head end of the list.

Parameters

in	*list*	The list that we want to pop (from head)

4.7.2.8 void listPush (**linkedList** ∗ *list,* void ∗ *value*)

Adds a new node containing data ∗value at the head of the list.

Parameters

in	*list*	The list to which we want to add
in	*value*	The value that we want to put in the new node (added at head).

4.7.2.9 void∗ listRemove (**linkedList** ∗ *list,* **linkedListNode** ∗ *node*)

Removes a node from list , if the node is present in the list.

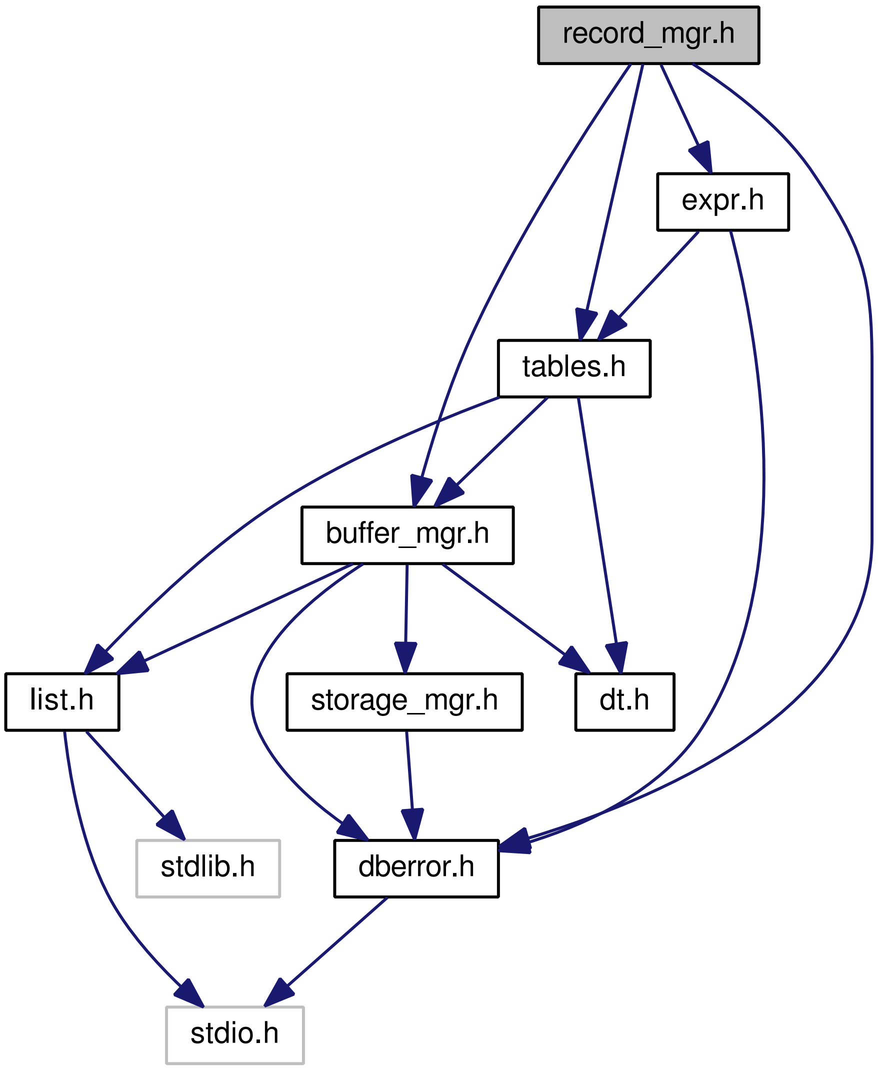
Parameters

in	*list*	The list from which we want to delete
in	*node*	The node that we want to remove from ∗list

4.8 record_mgr.h File Reference

```
#include "dberror.h"
#include "expr.h"
#include "tables.h"
#include "buffer_mgr.h"
```

Include dependency graph for record_mgr.h:



Classes

- struct **RM_ScanHandle**
- struct **RM_MetaScan**

Typedefs

- typedef struct **RM_ScanHandle RM_ScanHandle**
- typedef struct **RM_MetaScan RM_MetaScan**

Functions

- **RC initRecordManager** (void ∗mgmtData)
- **RC shutdownRecordManager** ()
- **RC createTable** (char ∗name, **Schema** ∗schema)
- **RC openTable** (**RM_TableData** ∗rel, char ∗name)
- **RC closeTable** (**RM_TableData** ∗rel)
- **RC deleteTable** (char ∗name)
- int **getNumTuples** (**RM_TableData** ∗rel)
- int **getNumPages** (**RM_TableData** ∗rel)
- int **getFirstFreePage** (**RM_TableData** ∗rel)
- char ∗ **createEmptyPage** (**Schema** ∗schema)
- **RC insertRecord** (**RM_TableData** ∗rel, **Record** ∗record)
- **RC deleteRecord** (**RM_TableData** ∗rel, **RID** id)
- **RC updateRecord** (**RM_TableData** ∗rel, **Record** ∗record)
- **RC getRecord** (**RM_TableData** ∗rel, **RID** id, **Record** ∗∗record)
- **RC startScan** (**RM_TableData** ∗rel, **RM_ScanHandle** ∗scan, **Expr** ∗cond)
- **RC next** (**RM_ScanHandle** ∗scan, **Record** ∗∗record)
- **RC closeScan** (**RM_ScanHandle** ∗scan)
- int **getRecordSize** (**Schema** ∗schema)
- int **getSerializedRecordSize** (**Schema** ∗schema)
- int **getNumRecordsPerPage** (**Schema** ∗schema)
- **Schema** ∗ **createSchema** (int numAttr, char ∗∗attrNames, **DataType** ∗dataTypes, int ∗typeLength, int keySize, int ∗keys)
- **RC freeSchema** (**Schema** ∗schema)
- **RC createRecord** (**Record** ∗∗record, **Schema** ∗schema)
- **RC freeRecord** (**Record** ∗record)
- **RC getAttr** (**Record** ∗record, **Schema** ∗schema, int attrNum, **Value** ∗∗value)
- **RC setAttr** (**Record** ∗record, **Schema** ∗schema, int attrNum, **Value** ∗value)

4.8.1 Typedef Documentation

4.8.1.1 typedef struct RM_MetaScan RM_MetaScan

This structure stores the meta data of a scan handle.

4.8.1.2 typedef struct RM_ScanHandle RM_ScanHandle

This structure does the book keeping for scan operations.

The mgmtData stores an **RM_MetaScan** (p. 22), that is mainly responsible for the meta data.

4.8.2 Function Documentation

4.8.2.1 RC closeScan (RM_ScanHandle ∗ *scan*)

This function marks the end of a scan operation.

It deallocates the mgmtData of the scan handle.

Parameters

`in`	*scan*	The scan handle whose meta data is to be deallocated.

4.8.2.2 RC closeTable (RM_TableData ∗ *rel*)

This function de allocates the the mgmtData and schema of rel, and disconects it to the table filename by shutting down the buffer_manager contained in mgmt data.

Parameters

`in`	*rel*	The table that we want to close e

4.8.2.3 char∗ createEmptyPage (Schema ∗ *schema*)

This function creates an empty data page for a table with given schema.

It writes the management and free space information on first TABLE_META_SIZE bytes of this page, and leaves the rest blank.

Parameters

`in`	*schema*	The schema of the table for which we are creating a new data page

4.8.2.4 RC createRecord (Record ∗∗ *record,* Schema ∗ *schema*)

This function allocates memory for a new record.

Parameters

`in`	*record*	The pointer to pointer of a record that is being allocated.
`in`	*schema*	The schema of the record.

4.8.2.5 Schema∗ createSchema (int *numAttr,* char ∗∗ *attrNames,* DataType ∗ *dataTypes,* int ∗ *typeLength,* int *keySize,* int ∗ *keys*)

This function creates a schema variable from the supplied inputs.

Parameters

`in`	*numAttr*	Number of Attributes in the schema
`in`	*attrNames*	An array of size numAttr containing names (char∗) of each attribute, passed as pointer to pointer to char
`in`	*dataTypes*	An array of size numAttr containing datatype of each attribute, passed as pointer to DataType
`in`	*typeLength*	An array of size numAttr containing size of each attribute's DataType, passed as pointer to DataType
`in`	*keySize*	Number of Keys for the table described by the schema
`in`	*dataTypes*	An array of size keySize containing index of each key attribute in attrNames array, passed as pointer to int

4.8.2.6 RC createTable (char ∗ *name,* Schema ∗ *schema*)

This function creates a new table .

Creates its underlying pagefile. writes the table info on first page, creates second page which is an empty dataPage. On first page (table info), the firstFreePage is initialized to 1 (pointing to the empty data page).

Parameters

in	*name*	Name of the new table
in	*schema*	**Schema** (p. 27) of the new table.

4.8.2.7 RC deleteRecord (RM_TableData ∗ *rel*, RID *id*)

This function is used to delete a record from table.

The record is found using id. If the record is not present, it returns a RC_RM_RECORD_NOT_FOUND.

Parameters

in	*rel*	The table from which we want to delete a record
in	*id*	**RID** (p. 22) =(page,slot) of the record which we want to delete.

4.8.2.8 RC deleteTable (char ∗ *name*)

This function destroys the underlying pageFile.

Parameters

in	*rel*	The table that we are trying to delete

4.8.2.9 RC freeRecord (Record ∗ *record*)

This function deallocates all memory allotted to a record variable.

Parameters

in	*record*	The record to be freed

4.8.2.10 RC freeSchema (Schema ∗ *schema*)

This function deallocates all memory allotted to a schema.

Parameters

in	*schema*	The schema to be freed

4.8.2.11 RC getAttr (Record ∗ *record*, Schema ∗ *schema*, int *attrNum*, Value ∗∗ *value*)

This function gets value of a particular attribute in a record.

Parameters

in	*record*	The record whose attribute is to be fetched
in	*schema*	The schema of the record whose attribute is to fetched
in	*attrNum*	The attribute number of the record which is to fetched
out	*value*	The value read from the record's attribute field

4.8.2.12 int getFirstFreePage (RM_TableData ∗ *rel*)

This function returns the page number of the first page with a free slot in the table.

A free page is created using createEmptyPage or as a result of deleting a record in a full page. Whenever a new free page is created (using either methods), the firstFreePage in tableInfo is assigned to nextFreePage on the new freePage. The firstFreePage in table info is then updated to the page number of the new freePage. With this scheme I create a symbolic linked list of free pages in the table with its head indicated by the firstFreePage in table info stored on first page.

Parameters

in	*rel*	The table in which we are looking for the first free page.

4.8.2.13 int getNumPages (**RM_TableData** ∗ *rel*)

This function returns the total number of data pages in a table.

Parameters

in	*rel*	The table in which we are looking for pages

4.8.2.14 int getNumRecordsPerPage (**Schema** ∗ *schema*)

This function determines how many records we want to write to a given page.

It computes the size as (PAGE_SIZE - TABLE_META_SIZE)/(size of serialized record);

Parameters

in	*schema*	The schema of the record for which we are computing this number.

4.8.2.15 int getNumTuples (**RM_TableData** ∗ *rel*)

This function returns the total number of records in a table.

Parameters

in	*rel*	The table in which we are looking for records

4.8.2.16 RC getRecord (**RM_TableData** ∗ *rel,* **RID** *id,* **Record** ∗∗ *record*)

This function fetches a record from a table.

The record is identified using its **RID** (p. 22), and the result is returned in record. This function returns RC_RM_RECORD_NOT_FOUND if the provided **RID** (p. 22) is empty.

Parameters

in	*rel*	The table from which we are trying to fetch the record.
in	***RID*** *(p. 22)*	The id<page,slot> of the record that we are trying to fetch
out	*record*	This is where the record from table is returned (if found, else its is left unallocated)

4.8.2.17 int getRecordSize (**Schema** ∗ *schema*)

This function computes the size of a non serialized version of the record.

Parameters

`in`	*schema*	The schema of the record for which we are computing the size.

4.8.2.18 int getSerializedRecordSize (**Schema** ∗ *schema*)

This function computes the size of a record in its serialized string representation.

Since data in dataPages are to be written to disk, it needs to be written in serialized form, so that we can read it back when needed. This size computation helps us determine how many records we want to write in a dataPage (if a fixed PAGE_SiZE). A serialized record looks like this : [1-0] (a:0,b:aaaa,c:3)

Parameters

`in`	*schema*	The schema of the record for which we are computing the size.

4.8.2.19 RC initRecordManager (void ∗ *mgmtData*)

This function initilizes a record manager.

I didn't find any practical use for it

4.8.2.20 RC insertRecord (**RM_TableData** ∗ *rel,* **Record** ∗ *record*)

This function is used to return a new record in the table.

The first empty (page, slot) is found and the record is added, and the id of the record is updated to this (page,slot). If there are no such empty (page,slots) then a new page is created where the record is entered (and the first free page in table info on first free page is updated to this new page)

Parameters

`in`	*rel*	The table in which we want to insert
`in,out`	*record*	The record which want to insert in the table rel. The **RID** (p. 22) where the record goes is updated in record->id.

4.8.2.21 RC next (**RM_ScanHandle** ∗ *scan,* **Record** ∗∗ *record*)

This function acts like an iterator on each record of the table.

After initilization of scan handle using startScan, every time this function is called, the next record in table, that satisfies scan->**Expr** (p. 15) condition, is returned. The **RID** (p. 22) of this record is updated in scan->mgmtData using the **RM_MetaScan** (p. 22) struct. The next time this function is called, iteration begins from this point onwards.

Parameters

`in`	*scan*	The scan handle that we are iterating on.
`out`	*record*	Stores the next record satisfying the scan conditions

4.8.2.22 RC openTable (**RM_TableData** ∗ *rel,* char ∗ *name*)

This function opens a table with tablename name.

In the process it allocates memory to mgmtData and initializes a buffer_manager of size RM_BUFFER_SIZE.

Parameters

out	*rel*	The name, schema and buffer_manager for the table is stored here
in	*name*	The name of the table that we want to open.

4.8.2.23 RC setAttr (Record ∗ *record,* Schema ∗ *schema,* int *attrNum,* Value ∗ *value*)

This function sets value of a particular attribute in a record.

Parameters

in	*record*	The record whose attribute is to saved
in	*schema*	The schema of the record whose attribute is to saved
in	*attrNum*	The attribute number of the record which is to saved
in	*value*	The new value to be set in the record's attribute field

4.8.2.24 RC shutdownRecordManager ( )

This function shuts down a record manager.

I didn't find any practical use for it

4.8.2.25 RC startScan (RM_TableData ∗ *rel,* RM_ScanHandle ∗ *scan,* Expr ∗ *cond*)

This function marks the begining of a scan procedure.

It allocates the mgmtData of scan handle and initializes it using information from **RM_TableData** (p. 26)

Parameters

in	*rel*	The table that we want to scan
in	*scan*	The scan handle that we want to use for scanning rel
in	***Expr*** *(p. 15)*	The condition that any record should satisfy to qualify as a result of this scan. If null, all records qualify

4.8.2.26 RC updateRecord (RM_TableData ∗ *rel,* Record ∗ *record*)

This function updates a record in the table.

The record is found using record->id. If record->id points to an empty slot, a RC_RM_RECORD_NOT_FOUND error is returned. I am sticking with (update semantics) here and not trying to do an (update or insert)

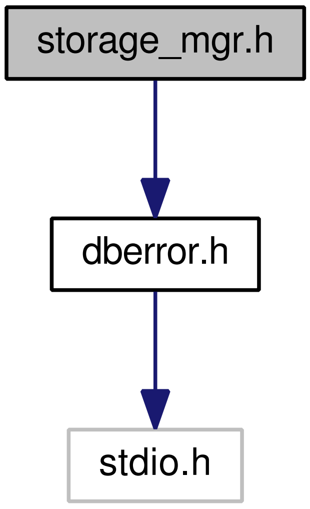
Parameters

in	*rel*	The table in which we are updating a record.
in	*record*	The record that we are trying to update.

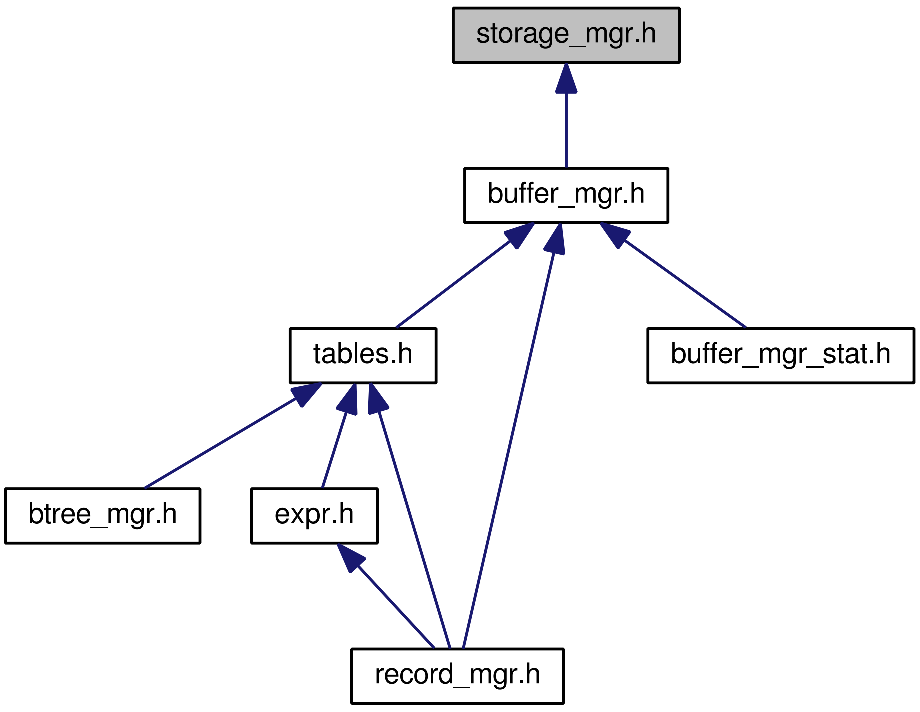
4.9 storage_mgr.h File Reference

```
#include "dberror.h"
```

Include dependency graph for storage_mgr.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct **SM_FileHandle**

Typedefs

- typedef struct **SM_FileHandle SM_FileHandle**
- typedef char ∗ **SM_PageHandle**

Functions

- void **initStorageManager** (void)
- **RC createPageFile** (char ∗fileName)
- **RC openPageFile** (char ∗fileName, **SM_FileHandle** ∗fHandle)
- **RC closePageFile** (**SM_FileHandle** ∗fHandle)
- **RC destroyPageFile** (char ∗fileName)
- **RC readBlock** (int pageNum, **SM_FileHandle** ∗fHandle, **SM_PageHandle** memPage)
- int **getBlockPos** (**SM_FileHandle** ∗fHandle)
- **RC readFirstBlock** (**SM_FileHandle** ∗fHandle, **SM_PageHandle** memPage)
- **RC readNextBlock** (**SM_FileHandle** ∗fHandle, **SM_PageHandle** memPage)
- **RC readPreviousBlock** (**SM_FileHandle** ∗fHandle, **SM_PageHandle** memPage)
- **RC readCurrentBlock** (**SM_FileHandle** ∗fHandle, **SM_PageHandle** memPage)
- **RC readLastBlock** (**SM_FileHandle** ∗fHandle, **SM_PageHandle** memPage)
- **RC writeBlock** (int pageNum, **SM_FileHandle** ∗fHandle, **SM_PageHandle** memPage)
- **RC writeCurrentBlock** (**SM_FileHandle** ∗fHandle, **SM_PageHandle** memPage)
- **RC appendEmptyBlock** (**SM_FileHandle** ∗fHandle)
- **RC ensureCapacity** (int numberOfPages, **SM_FileHandle** ∗fHandle)

4.9.1 Typedef Documentation

4.9.1.1 typedef struct SM_FileHandle SM_FileHandle

4.9.1.2 typedef char∗ SM_PageHandle

Definition at line 16 of file storage_mgr.h.

4.9.2 Function Documentation

4.9.2.1 RC appendEmptyBlock (SM_FileHandle ∗ *fHandle*)

4.9.2.2 RC closePageFile (SM_FileHandle ∗ *fHandle*)

4.9.2.3 RC createPageFile (char ∗ *fileName*)

4.9.2.4 RC destroyPageFile (char ∗ *fileName*)

4.9.2.5 RC ensureCapacity (int *numberOfPages,* SM_FileHandle ∗ *fHandle*)

4.9.2.6 int getBlockPos (SM_FileHandle ∗ *fHandle*)

4.9.2.7 void initStorageManager (void)

4.9.2.8 RC openPageFile (char ∗ *fileName,* SM_FileHandle ∗ *fHandle*)

4.9.2.9 RC readBlock (int *pageNum,* SM_FileHandle ∗ *fHandle,* SM_PageHandle *memPage*)

4.9.2.10 RC readCurrentBlock (SM_FileHandle ∗ *fHandle,* SM_PageHandle *memPage*)

4.9.2.11 RC readFirstBlock (SM_FileHandle ∗ *fHandle,* SM_PageHandle *memPage*)

4.9.2.12 RC readLastBlock (SM_FileHandle ∗ *fHandle,* SM_PageHandle *memPage*)

4.9.2.13 RC readNextBlock (SM_FileHandle ∗ *fHandle,* SM_PageHandle *memPage*)

4.9.2.14 RC readPreviousBlock (SM_FileHandle ∗ *fHandle*, SM_PageHandle *memPage*)

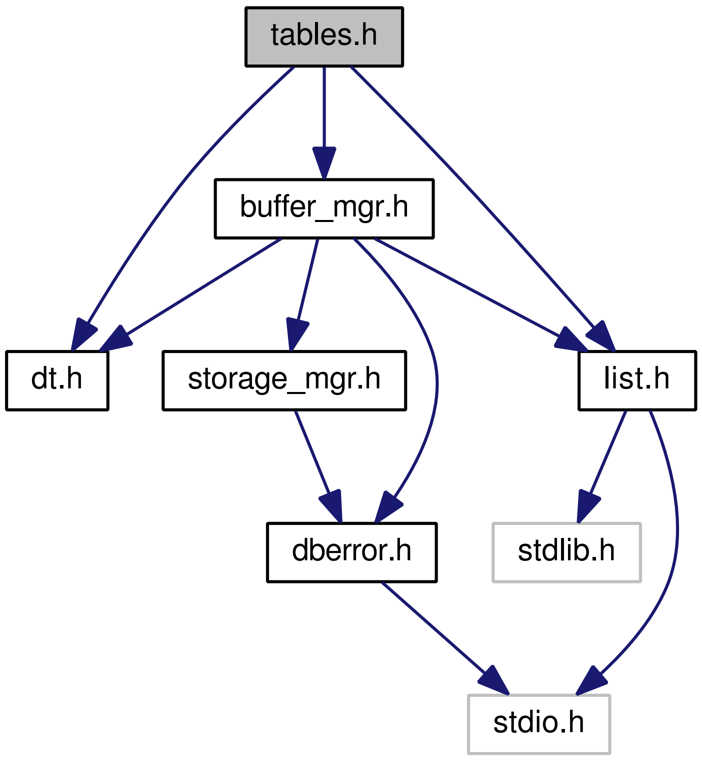
4.9.2.15 RC writeBlock (int *pageNum*, SM_FileHandle ∗ *fHandle*, SM_PageHandle *memPage*)

4.9.2.16 RC writeCurrentBlock (SM_FileHandle ∗ *fHandle*, SM_PageHandle *memPage*)

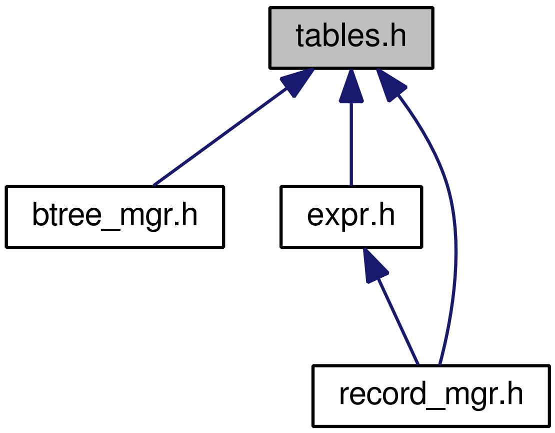
4.10 tables.h File Reference

```
#include "dt.h"
#include "list.h"
#include "buffer_mgr.h"
```

Include dependency graph for tables.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct **Value**
- union **Value::v**
- struct **RID**
- struct **Record**
- struct **Schema**
- struct **RM_TableData**
- struct **RM_RelData**
- struct **VarString**

Macros

- #define **MAKE_VARSTRING**(var)
- #define **FREE_VARSTRING**(var)
- #define **GET_STRING**(result, var)
- #define **RETURN_STRING**(var)
- #define **ENSURE_SIZE**(var, newsize)
- #define **APPEND_STRING**(var, string)
- #define **APPEND**(var,...)
- #define **MAKE_STRING_VALUE**(result, value)
- #define **MAKE_VALUE**(result, datatype, value)

Typedefs

- typedef enum **DataType DataType**
- typedef struct **Value Value**
- typedef struct **RID RID**
- typedef struct **Record Record**
- typedef struct **Schema Schema**
- typedef struct **RM_TableData RM_TableData**
- typedef struct **RM_RelData RM_RelData**
- typedef struct **VarString VarString**

Enumerations

- enum **DataType** { **DT_INT** = 0, **DT_STRING** = 1, **DT_FLOAT** = 2, **DT_BOOL** = 3 }

Functions

- char ∗ **serializeTableInfo** (**RM_TableData** ∗rel)
- char ∗ **serializeTableContent** (**RM_TableData** ∗rel)
- char ∗ **serializeSchema** (**Schema** ∗schema)
- char ∗ **serializeRecord** (**Record** ∗record, **Schema** ∗schema)
- char ∗ **serializeAttr** (**Record** ∗record, **Schema** ∗schema, int attrNum)
- char ∗ **serializeValue** (**Value** ∗val)
- **RC attrOffset** (**Schema** ∗schema, int attrNum, int ∗result)
- **Value** ∗ **stringToValue** (char ∗value)
- **Schema** ∗ **stringToSchema** (char ∗schemaLine)
- **Schema** ∗ **stringToTabInfo** (char ∗tabInfo, char ∗∗name, int ∗numTuples, int ∗numPages, int ∗firstFreePage)
- **Record** ∗ **stringToRecord** (char ∗recordStr, **Schema** ∗schema)
- void **freeValue** (**Value** ∗val)
- char ∗∗ **strsplit** (char ∗str, char ∗delim, int ∗numSplit)
- void **freeStrArr** (char ∗∗str, int cnt)

4.10.1 Macro Definition Documentation

4.10.1.1 #define APPEND(*var, ...*)

Value:

```
do {                            \
    char *tmp = malloc(10000);            \
    sprintf(tmp, __VA_ARGS__);                APPEND_STRING(var,tmp);          \
    free(tmp);                      \
  } while(0)
```

Definition at line 175 of file tables.h.

4.10.1.2 #define APPEND_STRING(*var, string*)

Value:

```
do {                                        ENSURE_SIZE(var, var->size + strlen(string));           \
    memcpy(var->buf + var->size, string, strlen(string));       \
    var->size += strlen(string);                    \
  } while(0)
```

Definition at line 168 of file tables.h.

4.10.1.3 #define ENSURE_SIZE(*var, newsize*)

Value:

```
do {                                    \
    if (var->bufsize < newsize)                 \
    {                                   \
      int newbufsize = var->bufsize;                \
      while((newbufsize *= 2) < newsize);           \
      var->buf = realloc(var->buf, newbufsize);         \
    }                                   \
  } while (0)
```

Definition at line 158 of file tables.h.

4.10.1.4 #define FREE_VARSTRING(*var*)

Value:

```
do {                            \
  free(var->buf);                   \
  free(var);                        \
  } while (0)
```

Definition at line 137 of file tables.h.

4.10.1.5 #define GET_STRING(*result, var*)

Value:

```
do {                            \
    result = malloc((var->size) + 1);       \
    memcpy(result, var->buf, var->size);    \
    result[var->size] = '\0';           \
  } while (0)
```

Definition at line 143 of file tables.h.

4.10.1.6 #define MAKE_STRING_VALUE(*result,* *value*)

Value:

```
do {                                        \
    (result) = (Value *) malloc(sizeof(Value));             \
    (result)->dt = DT_STRING;                       \
    (result)->v.stringV = (char *) malloc(strlen(value) + 1);       \
    strcpy((result)->v.stringV, value);                 \
  } while(0)
```

Definition at line 185 of file tables.h.

4.10.1.7 #define MAKE_VALUE(*result,* *datatype,* *value*)

Value:

```
do {                                        \
    (result) = (Value *) malloc(sizeof(Value));             \
    (result)->dt = datatype;                        \
    switch(datatype)                            \
      {                                     \
      case DT_INT:                              \
    (result)->v.intV = value;                       \
    break;                                  \
      case DT_FLOAT:                                \
    (result)->v.floatV = value;                     \
    break;                                  \
      case DT_BOOL:                             \
    (result)->v.boolV = value;                      \
    break;                                  \
      }                                     \
  } while(0)
```

Definition at line 194 of file tables.h.

4.10.1.8 #define MAKE_VARSTRING(*var*)

Value:

```
do {                                \
  var = (VarString *) malloc(sizeof(VarString));    \
  var->size = 0;                        \
  var->bufsize = 100;                       \
  var->buf = malloc(100);                   \
  } while (0)
```

Definition at line 129 of file tables.h.

4.10.1.9 #define RETURN_STRING(*var*)

Value:

```
do {                        \
    char *resultStr;                    GET_STRING(resultStr, var);             FREE_VARSTRING(var);                \
    return resultStr;               \
  } while (0)
```

Definition at line 150 of file tables.h.

4.10.2 Typedef Documentation

4.10.2.1 typedef enum DataType DataType

An enum to capture different Data Types supported for colums of tables.

4.10.2.2 typedef struct Record Record

A record is any one row of our table.

It has a **RID** (p. 22) id identifier and corresponding data

4.10.2.3 typedef struct RID RID

Serves as unique id of each record written to the table, and stores the relative position of a record within the table.

4.10.2.4 typedef struct RM_RelData RM_RelData

RelData would be responsible for tracking and managing data and free space of table data.

I have assumed that for each table, the first page contains schema and some meta info. Second page onwards we have page data. In page data I store 2 things before records. 1) a boolean array of size numRecordsPerPage indicating whether the corresponding slot is empty or full. 2) the page number of next free page

4.10.2.5 typedef struct RM_TableData RM_TableData

TableData: Management Structure for a **Record** (p. 21) Manager to handle one relation.

4.10.2.6 typedef struct Schema Schema

Schema (p. 27) contains information of a table schema: its attributes, datatypes, sizes of each datatype, key attributes and number of keys.

4.10.2.7 typedef struct Value Value

Value (p. 29) stores data of any one cell <recordNumber, attrNum>.

The data can be of any one of the types defined in enum DataType

4.10.2.8 typedef struct VarString VarString

dynamic string

4.10.3 Enumeration Type Documentation

4.10.3.1 enum DataType

An enum to capture different Data Types supported for colums of tables.

Enumerator

DT_INT

DT_STRING

DT_FLOAT

DT_BOOL

Definition at line 12 of file tables.h.

4.10.4 Function Documentation

4.10.4.1 RC attrOffset (Schema ∗ *schema,* int *attrNum,* int ∗ *result*)

This function helps computing the offset of an attribute in record->data given its schema.

Parameters

in	*schema*	The schema for which we are computing attribute offset in records
in	*attrNum*	The schema contains many attributes. attrNum specifies one of them.
out	*result*	Contains the memory offset computed for the specified attrNum

4.10.4.2 void freeStrArr (char ∗∗ *str,* int *cnt*)

A utility function for freeing an array of strings of length cnt;.

Parameters

in	*str*	string array passed as pointer to pointer to char
in	*cnt*	the number of elements to be freed

4.10.4.3 void freeValue (Value ∗ *val*)

A utility function to free the memory allocated to a **Value** (p. 29) variable.

A simple free(value) is not enough because it contains a char ∗buff which might have a malloc allocation. This was one the sources of memory leaks in the supplied test cases.

Parameters

in	*val*	The value to be freed

4.10.4.4 char∗ serializeAttr (Record ∗ *record,* Schema ∗ *schema,* int *attrNum*)

This function serializes an individual attribute (column) of a record of a table.

Parameters

in	*record*	The record for which we are serializing a column.
in	*schema*	The schema of the record (needed for offset computation)
in	*attrNum*	specifies the attribute that we want to serialize

4.10.4.5 char∗ serializeRecord (Record ∗ *record,* Schema ∗ *schema*)

This function serializes an individual record and converts it to an equivalent string representation.

Parameters

in	*record*	The record to be serialized
in	*schema*	The schema of the record

4.10.4.6 char∗ serializeSchema (Schema ∗ *schema*)

This function serializes a **Schema** (p. 27) variable and converts it to an equivalent string representation This function serializes an individual record and converts it to an equivalent string representation.

Parameters

in	*schema*	The schema to be serialized

4.10.4.7 char∗ serializeTableContent (RM_TableData ∗ *rel*)

This function serializes all the records of a table into one big string.

This function is only good for debugging purposes

Parameters

in	*rel*	The table for which we want to serialize the table content

4.10.4.8 char∗ serializeTableInfo (RM_TableData ∗ *rel*)

This function serializes the table info part of an **RM_TableData** (p. 26) to string.

The table info refers to the first page on disk for each table, which contains meta information for the table like its schema, first free page etc.

Parameters

in	*rel*	The table for which we want to serialize the table info

4.10.4.9 char∗ serializeValue (Value ∗ *val*)

This function serializes an individual **Value** (p. 29) variable.

Returns string

Parameters

in	*val*	The value to be serialized

4.10.4.10 Record∗ stringToRecord (char ∗ *recordStr,* Schema ∗ *schema*)

This function converts a string variable to corresponding record.

Parameters

in	*recordStr*	The serialized char∗ representation of a record
in	*schema*	The schema according to which the record is to be parsed

4.10.4.11 Schema∗ stringToSchema (char ∗ *schemaLine*)

This function takes in a serialized reprensetation of schema of a table, and parses out all relevant information contained and returns the schema.

Parameters

in	*schemaLine*	The serialized string representation of a schema.

4.10.4.12 Schema∗ stringToTabInfo (char ∗ *tabInfo,* char ∗∗ *name,* int ∗ *numTuples,* int ∗ *numPages,* int ∗ *firstFreePage*)

This function takes in a serialized reprensetation of the first meta information of a table, and parses out all relevant information contained in this page.

It returns the schema.

Parameters

in	*tabInfo*	Contains the string in first page of a table
out	*name*	The name of table as read out from first page
out	*numTuples*	The total number of records stored in table as read out from first page.
out	*numPages*	The total number of data pages in table as read out from first page.
out	*firstFreePage*	The number of first page with free slots in table as read out from first page

4.10.4.13 Value∗ stringToValue (char ∗ *value*)

Creates **Value** (p. 29) from a string containing a serialized representation of a value.

Parameters

in	*value*	The string containing a serialized representation of a **Value** (p. 29) .

4.10.4.14 char∗∗ strsplit (char ∗ *str,* char ∗ *delim,* int ∗ *numSplit*)

A utility function for splitting a string on multiple delimitters.

Returns an array of substrings split on delim

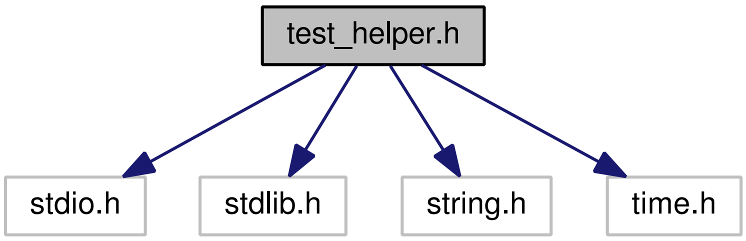
Parameters

in	*str*	The string to be split
in	*delim*	The array of delimitters passed as char ∗
out	*numSplit*	The number of components in the resultant split

4.11 test_helper.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
```

Include dependency graph for test_helper.h:



Macros

- #define **TEST_INFO** __FILE__, **testName**, __LINE__, __TIME__
- #define **TEST_CHECK**(code)
- #define **ASSERT_EQUALS_STRING**(expected, real, message)
- #define **ASSERT_EQUALS_INT**(expected, real, message)
- #define **ASSERT_TRUE**(real, message)
- #define **ASSERT_ERROR**(expected, message)
- #define **TEST_DONE**()

Variables

- char ∗ **testName**

4.11.1 Macro Definition Documentation

4.11.1.1 #define ASSERT_EQUALS_INT(*expected, real, message*)

Value:

```
do {                                    \
    if ((expected) != (real))                   \
      {                                 \
    printf("[%s-%s-L%i-%s] FAILED: expected <%i> but was <%i>: %s\n",TEST_INFO, expected, real, message); \
    exit(1);                            \
      }                                 \
    printf("[%s-%s-L%i-%s] OK: expected <%i> and was <%i>: %s\n",TEST_INFO, expected, real, message); \
  } while(0)
```

Definition at line 40 of file test_helper.h.

4.11.1.2 #define ASSERT_EQUALS_STRING(*expected, real, message*)

Value:

```
do {                                    \
    if (strcmp((expected),(real)) != 0)                 \
      {                                 \
    printf("[%s-%s-L%i-%s] FAILED: expected <%s> but was <%s>: %s\n",TEST_INFO, expected, real, message); \
    exit(1);                            \
      }                                 \
    printf("[%s-%s-L%i-%s] OK: expected <%s> and was <%s>: %s\n",TEST_INFO, expected, real, message); \
  } while(0)
```

Definition at line 29 of file test_helper.h.

4.11.1.3 #define ASSERT_ERROR(*expected, message*)

Value:

```
do {                                    \
    int result = (expected);                        \
    if (result == (RC_OK))                      \
      {                                 \
    printf("[%s-%s-L%i-%s] FAILED: expected an error: %s\n",TEST_INFO, message); \
    exit(1);                            \
      }                                 \
    printf("[%s-%s-L%i-%s] OK: expected an error and was RC <%i>: %s\n",
      TEST_INFO,  result , message); \
  } while(0)
```

Definition at line 63 of file test_helper.h.

4.11.1.4 #define ASSERT_TRUE(*real, message*)

Value:

```
do {                                    \
    if (!(real))                            \
      {                                 \
    printf("[%s-%s-L%i-%s] FAILED: expected true: %s\n",TEST_INFO, message); \
    exit(1);                            \
      }                                 \
    printf("[%s-%s-L%i-%s] OK: expected true: %s\n",TEST_INFO, message); \
  } while(0)
```

Definition at line 51 of file test_helper.h.

4.11.1.5 #define TEST_CHECK(*code*)

Value:

```
do {                                    \
    int rc_internal = (code);                       \
    if (rc_internal != RC_OK)                       \
      {                                 \
    char *message = errorMessage(rc_internal);          \
    printf("[%s-%s-L%i-%s] FAILED: Operation returned error: %s\n",TEST_INFO, message); \
    free(message);                          \
    exit(1);                            \
      }                                 \
  } while(0);
```

Definition at line 16 of file test_helper.h.

4.11.1.6 #define TEST_DONE( )

Value:

```
do {                                    \
    printf("[%s-%s-L%i-%s] OK: finished test\n\n",TEST_INFO); \
  } while (0)
```

Definition at line 75 of file test_helper.h.

4.11.1.7 #define TEST_INFO __FILE__, testName, __LINE__, __TIME__

Definition at line 13 of file test_helper.h.

4.11.2 Variable Documentation

4.11.2.1 char∗ testName

Index