

Jialuo Gao & Shawn Zhong & Suyan Qu  
CS 564: Database Management Systems  
PP 3: BTree Index  
April 14, 2019

### **Group Member**

- Shawn Zhong:
  - CS Login: szhong
  - Net ID: wzhong36
- Suyan Qu:
  - CS Login: suyan
  - Net ID: squ27
- Jialuo Gao:
  - CS Login: jialuo
  - Net ID: jgao223

### **How to run**

- `make && ./src/badgerdb_main`

### **Assumptions**

- All valid page numbers are nonzero.
- All valid records are nonzero.
- No duplicate keys are inserted.
- All records in a file have the same length.
- B+ Tree only supports single-attribute indexing
- The indexed attributes are all integer data type

## Implementation Choices

- The level field in the node is used to distinguish between leaf nodes and internal nodes. For efficiency, internal nodes all have level 0 while all leaf nodes have level -1.
- Each key in the internal node stores the smallest key in the rightmost leaf node stored in the subtree pointed to by the pointer to the right of that key in the internal node. In another word, in an internal node, all key-record pairs stored in the subtree to the left of a key are smaller than that key while all key-record pairs stored in the subtree to the right will be greater than or equal to that key.

## Implementation for Duplicate Keys

- Allowing duplicate keys would lead to some different behaviors when a node splits. If the key at index where split occurs has a duplicate, not only that key but also its duplicate need to be moved to a newly created node.
- When initing scan, our current implementation finds the first occurrence of the lower bound. And if the `lowOp` is set to `GT` (greater than), we move to the next entry. If duplicates are allowed, the next entry may very likely have the same key, so it is necessary to continuously move to the next entry until we find a key that is greater than the lower bound. This can be very inefficient if there are many duplicated keys.

## Buffer Management (When to Pin & Unpin)

- During recursive insertion, a page for the current internal node is retrieved to find which node in the next level should be visited next. When returning from the next level, it is decided if the child node has split and if the current node needs a split. If neither case occurs, the page is unpinned with dirty bit set to false. If the child node has split and the current node does not need to split, the current node is overwritten and then unpinned with dirty bit set to true. If the current node requires a split, a new page is allocated, information stored in the second half of the current node is cleared and moved to the newly created node, then pages for both nodes are overwritten and unpinned with dirty bit set to true. Similarly, a page for the leaf node is retrieved to find the insertion location. After insertion, the page for the current node is overwritten and unpinned with dirty bit set to true. If the leaf node is full and split is required, the page for the newly created node is overwritten and unpinned with dirty bit set to true simultaneously.
- When initializing scan, we recursively find the page containing the given lower bound. At an internal node, the page is retrieved to find the page number we should move to next, and is unpinned before moving to the next page. If we reach page storing a leaf node, this page is kept pinned.
- During scanning, we continuously read the pinned leaf page until we have returned the last element stored in this page or the current entry is greater than the upper bound given. If we reach the end of this pinned leaf page, the next page pointed to by this page is pinned and the original page is unpinned. If we reach an element with a key greater than the upper bound, then the scan has complete and the pinned page is unpinned.
- If end scan is explicitly called, the currently pinned page is also unpinned.

## Runtime Analysis

We used `test3_contiguous_random()` with `relationSize = 5000` to generate the profile report below.

▼ 3,798 samples * 100.00% of parent * 100.00% of all	PP3`0x1
▼ 3,798 samples * 100.00% of parent * 100.00% of all	libdyld.dylib`start
▼ 3,798 samples * 100.00% of parent * 100.00% of all	PP3`main
▼ 3,790 samples * 99.79% of parent * 99.79% of all	PP3`test3_contiguous_random
▶ 3,463 samples * 91.37% of parent * 91.18% of all	PP3`createRelationRandom
▼ 324 samples * 8.55% of parent * 8.53% of all	PP3`intTests
▼ 301 samples * 92.90% of parent * 7.93% of all	PP3`badgerdb::BTreeIndex::BTreeIndex
▼ 300 samples * 99.67% of parent * 7.90% of all	PP3`badgerdb::BTreeIndex::BTreeIndex
▼ 196 samples * 65.33% of parent * 5.16% of all	PP3`badgerdb::BTreeIndex::insertEntry
▼ 196 samples * 100.00% of parent * 5.16% of all	PP3`badgerdb::BTreeIndex::insert
▶ 172 samples * 87.76% of parent * 4.53% of all	PP3`badgerdb::BTreeIndex::insert
▶ 16 samples * 8.16% of parent * 0.42% of all	PP3`badgerdb::findIndexNonLeaf
▶ 3 samples * 1.53% of parent * 0.08% of all	PP3`badgerdb::BufMgr::unPinPage
▶ 2 samples * 1.02% of parent * 0.05% of all	PP3`badgerdb::BufMgr::readPage
▶ 89 samples * 29.67% of parent * 2.34% of all	PP3`badgerdb::FileScan::scanNext
▶ 11 samples * 3.67% of parent * 0.29% of all	PP3`badgerdb::FileScan::getRecord
3 samples * 1.00% of parent * 0.08% of all	libsystem_malloc.dylib`free_tiny
▶ 23 samples * 7.10% of parent * 0.61% of all	PP3`intScan

- Most of the time spent during the test is creating the relation, which takes 91.18% of the total time. The remaining 8.53% of the time is used for indexing and scanning.
- Indexing takes 5.16% of the total runtime. We insert the key-(record id) pair one by one to the B+ tree. On average, each insertion takes  $O(\log n)$  time, where  $n$  is the height of the B+ tree.
- Scanning takes 0.61% of the total runtime. Most of the scanning time is for initializing. Once we find the first key, we can just use the list structure of leaf nodes to search for the next key until we reach the desired value or upper bound.

## Tests

- **All tests are included in main.cpp, and all test cases are called by default**
- `test1_contiguous_ascending()`
  - Given test which creates key-value pairs from 0 to `relationSize` which is defaulted to be 5000 in ascending order
  - Test if the output amount matches the input amount when scanning
- `test2_contiguous_descending()`
  - Given test which creates key-value pairs from `relationSize` to zero which is defaulted to be 5000 in descending order
  - Test if the output amount matches the input amount when scanning
- `test3_contiguous_random()`
  - Given test which creates key-value pairs from 0 to `relationSize` which is defaulted to be 5000 in random order by shuffling the input vector
  - Test if the output amount matches the input amount when scanning
- `test4_out_of_bound()`
  - This test creates key-value pairs from 0 to `relationSize` which is defaulted to be 5000 in random order by shuffling the input vector
  - Test if the output amount matches the input amount when scanning
  - This test finds keys that are not all in the range of input
- `test5_noncontiguous_random()`
  - Insert randomly generated values in random order (no duplicates)
    - Create an ascending order vector of `negative relationSize` to `relationSize`
    - Each number has a 10% probability of getting picked

- Shuffle the vector
  - Test if the output amount matches the input amount when scanning
  - Test if outputs match the sorted input vector
- `test6_contiguous_ascending_stress()`
  - Creates key-value pairs from 0 to a huge `relationSize` in ascending order
  - Test if the output amount matches the input amount when scanning
- `test7_contiguous_descending_stress()`
  - Creates key-value pairs from 0 to a huge `relationSize` in descending order
  - Test if the output amount matches the input amount when scanning
- `test8_contiguous_random_stress()`
  - Creates key-value pairs from 0 to a huge `relationSize` in random order by shuffling the vector
  - Test if the output amount matches the input amount when scanning
- `test9_error_test()`
  - Test whether `ScanNotInitializedException` will be thrown if `endScan` is called before `startScan`
  - Test whether `ScanNotInitializedException` will be thrown if `scanNext` is called before `startScan`
  - Test whether `BadOpcodesException` will be thrown if `lowOp` is `LTE`
  - Test whether `BadOpcodesException` will be thrown if `highOp` is `GTE`
  - Test whether `BadOpcodesException` will be thrown if `lowValInt` > `highValInt`

# badgerdb::BTreeIndex Class Reference

**BTreeIndex** class. It implements a B+ Tree index on a single attribute of a relation. This index supports only one scan at a time. [More...](#)

```
#include <btree.h>
```

## Classes

```
struct indexMetaInfo
```

## Public Member Functions

```
BTreeIndex (const std::string &relationName, std::string &outIndexName, BufMgr *bufMgrIn, const int  
attrByteOffset, const Datatype attrType)
```

```
~BTreeIndex ()
```

```
const void insertEntry (const void *key, const RecordId rid)
```

```
const void startScan (const void *lowVal, const Operator lowOp, const void *highVal, const Operator highOp)
```

```
const void scanNext (RecordId &outRid)
```

```
const void endScan ()
```

## Private Member Functions

```
LeafNodeInt * allocLeafNode (PageId &newPageId)
```

```
NonLeafNodeInt * allocNonLeafNode (PageId &newPageId)
```

```
bool isLeaf (Page *page)
```

```
bool isNonLeafNodeFull (NonLeafNodeInt *node)
```

```
bool isLeafNodeFull (LeafNodeInt *node)
```

```
int getLeafLen (LeafNodeInt *node)
```

```
int getNonLeafLen (NonLeafNodeInt *node)
```

```
int findArrayIndex (const int *arr, int len, int key, bool includeKey=true)
```

```
int findIndexNonLeaf (NonLeafNodeInt *node, int key)
```

```
int findInsertionIndexLeaf (LeafNodeInt *node, int key)
```

```
int findScanIndexLeaf (LeafNodeInt *node, int key, bool includeKey)
```

```
void insertToLeafNode (LeafNodeInt *node, int i, int key, RecordId rid)
```

```
void insertToNonLeafNode (NonLeafNodeInt *n, int i, int key, PageId pid)
```

```
void splitLeafNode (LeafNodeInt *node, LeafNodeInt *newNode, int index)
```

```
void splitNonLeafNode (NonLeafNodeInt *curr, NonLeafNodeInt *next, int i, bool keepMidKey)
```

```
PageId splitRoot (int midVal, PageId pid1, PageId pid2)
```

```
PageId insertToLeafPage (Page *origPage, PageId origPageId, int key, RecordId rid, int &midVal)
```

```
PageId insert (PageId origPageId, int key, RecordId rid, int &midVal)
```

	void	<b>moveToNextPage</b>	( <b>LeafNodeInt</b> *node)
	void	<b>setPageIdForScan</b>	()
	void	<b>setEntryIndexForScan</b>	()
	void	<b>setNextEntry</b>	()

## Private Attributes

	File *	<b>file</b>	{}
	BufMgr *	<b>bufMgr</b>	{}
<b>Datatype</b>		<b>attributeType</b>	
	int	<b>attrByteOffset</b>	{}
	bool	<b>scanExecuting</b>	{}
	int	<b>nextEntry</b>	{}
	PageId	<b>currentPageNum</b>	{}
	Page *	<b>currentPageData</b>	{}
	int	<b>lowValInt</b>	{}
	int	<b>highValInt</b>	{}
<b>Operator</b>		<b>lowOp</b>	{GT}
<b>Operator</b>		<b>highOp</b>	{LT}

## Detailed Description

**BTreeIndex** class. It implements a B+ Tree index on a single attribute of a relation. This index supports only one scan at a time.

## Constructor & Destructor Documentation

◆ BTreeIndex()



```
badgerdb::BTreeIndex::BTreeIndex ( const std::string & relationName,
                                   std::string &      outIndexName,
                                   BufMgr *           bufMgrIn,
                                   const int          attrByteOffset,
                                   const Datatype      attrType
                                   )
```

**BTreeIndex** Constructor. Check to see if the corresponding index file exists. If so, open the file. If not, create it and insert entries for every tuple in the base relation using FileScan class.

#### Parameters

- relationName** Name of file.
- outIndexName** Return the name of index file.
- bufMgrIn** Buffer Manager Instance
- attrByteOffset** Offset of attribute, over which index is to be built, in the record
- attrType** Datatype of attribute over which index is built

#### Exceptions

- If the index file already exists for the corresponding attribute, but values in **BadIndexInfoException** metapage(relationName, attribute byte offset, attribute type etc.) do not match with values received through constructor parameters.

#### ◆ ~BTreeIndex()

```
badgerdb::BTreeIndex::~~BTreeIndex ( )
```

**BTreeIndex** Destructor. End any initialized scan, flush index file, after unpinning any pinned pages, from the buffer manager and delete file instance thereby closing the index file. Destructor should not throw any exceptions. All exceptions should be caught in here itself.

## Member Function Documentation

---

#### ◆ allocLeafNode()

**LeafNodeInt\*** badgerdb::BTreeIndex::allocLeafNode ( PageId & newPageId )

private

Alloc a page in the buffer for a leaf node

#### Parameters

**newPageId** the page number for the new node

#### Returns

a pointer to the new leaf node

### ◆ allocNonLeafNode()

**NonLeafNodeInt\*** badgerdb::BTreeIndex::allocNonLeafNode ( PageId & newPageId )

private

Alloca a page in the buffer for an internal node

#### Parameters

**newPageId** the page number for the new node

#### Returns

a pointer to the new internal node

### ◆ endScan()

const void badgerdb::BTreeIndex::endScan ( )

Terminate the current scan. Unpin any pinned pages. Reset scan specific variables.

#### Exceptions

**ScanNotInitializedException** If no scan has been initialized.

### ◆ findArrayIndex()

```
int badgerdb::BTreeIndex::findArrayIndex ( const int * arr,
                                           int      len,
                                           int      key,
                                           bool     includeKey = true
                                           )
```

private

Given an integer array, find the index of the first integer larger than (or equal to) the given key.

Assumption: The array is sorted.

### Parameters

**arr** an interger array  
**len** the length of the array  
**key** the target key  
**includeKey** whether the current key is included

### Returns

a. the index of the first integer larger than the given key if includeKey = false  
 b. the index of the first integer larger than or equal to the given key if includeKey = true  
 c. -1 if the key is not found till the end of array

## ◆ findIndexNonLeaf()

```
int badgerdb::BTreeIndex::findIndexNonLeaf ( NonLeafNodeInt * node,
                                              int              key
                                              )
```

private

Find the index of the first key smaller than the given key

Assumption:

1. All records are continuously stored.
2. All valid page numbers are nonzero.
3. All keys are sorted in the node.

### Parameters

**node** an internal node  
**key** the key to find

### Returns

the index of the first key smaller than the given key return the largest index if not found

## ◆ findInsertionIndexLeaf()

```
int badgerdb::BTreeIndex::findInsertionIndexLeaf ( LeafNodeInt * node,  
                                                    int             key  
                                                    )
```

private

Find the insertaion index for a key in a leaf node

Assumption:

1. All records are continuously stored.
2. All valid records are nonzero.
3. All keys are sorted in the node.

### Parameters

**node** a leaf node

**key** the key to be inserted

### Returns

the insertaion index for a key in a leaf node

## ◆ findScanIndexLeaf()

```
int badgerdb::BTreeIndex::findScanIndexLeaf ( LeafNodeInt * node,
                                              int          key,
                                              bool         includeKey
                                              )
```

private

Find the index of the first key larger than the given key in the leaf node

Assumption:

1. All records are continuously stored.
2. All valid records are nonzero.
3. All keys are sorted in the node.

#### Parameters

**node**            a leaf node  
**key**            the key to find  
**includeKey** whether the current key is included

#### Returns

a. the index of the first integer larger than the given key if includeKey = false  
 b. the index of the first integer larger than or equal to the given key if includeKey = true  
 c. -1 if the key is not found till the end of array

### ◆ getLeafLen()

```
int badgerdb::BTreeIndex::getLeafLen ( LeafNodeInt * node )
```

private

Returns the number of records stored in the leaf node.

Assumption:

1. All records are continuously stored.
2. All valid records are nonzero.

#### Parameters

**node** a leaf node

#### Returns

the number of records stored in the leaf node

### ◆ getNonLeafLen()

```
int badgerdb::BTreeIndex::getNonLeafLen ( NonLeafNodeInt * node )
```

private

Returns the number of records stored in the internal node.

Assumption:

1. All records are continuously stored.
2. All valid page numbers are nonzero.

#### Parameters

**node** an internal node

#### Returns

the number of records stored in the internal node

### ◆ insert()

```
Pageld badgerdb::BTreeIndex::insert ( Pageld  origPageld,  
                                     int      key,  
                                     RecordId rid,  
                                     int &    midVal  
                                     )
```

private

Recursively insert the given key-record pair into the subtree with the given root node. If the root node requires a split, the page number of the newly created node will be return

#### Parameters

**origPageld** page id of the page that stores the root node of the subtree.

**key** the key of the key-record pair to be inserted

**rid** the record ID of the key-record pair to be inserted

**midVal** a pointer to an integer value to be stored in the parent node. If the insertion requires a split in the current level, midVal is set to the smallest key stored in the subtree pointed by the newly created node.

#### Returns

the page number of the newly created node if a split occurs, or 0 otherwise.

### ◆ insertEntry()

```
const void badgerdb::BTreeIndex::insertEntry ( const void *    key,
                                              const RecordId rid
                                              )
```

Insert a new entry using the pair <value,rid>. Start from root to recursively find out the leaf to insert the entry in. The insertion may cause splitting of leaf node. This splitting will require addition of new leaf page number entry into the parent non-leaf, which may in-turn get split. This may continue all the way upto the root causing the root to get split. If root gets split, metapage needs to be changed accordingly. Make sure to unpin pages as soon as you can.

#### Parameters

**key** Key to insert, pointer to integer/double/char string

**rid** Record ID of a record whose entry is getting inserted into the index.

#### ◆ insertToLeafNode()

```
void badgerdb::BTreeIndex::insertToLeafNode ( LeafNodeInt * node,
                                              int             i,
                                              int             key,
                                              RecordId      rid
                                              )
```

private

Inserts the given key-record pair into the leaf node at the given insertion index.

#### Parameters

**node** a leaf node

**i** the insertion index

**key** the key of the key-record pair to be inserted

**rid** the record ID of the key-record pair to be inserted

#### ◆ insertToLeafPage()

```

PageId badgerdb::BTreeIndex::insertToLeafPage ( Page *   origPage,
                                                PageId   origPageId,
                                                int       key,
                                                RecordId  rid,
                                                int &    midVal
                                                )

```

private

Insert the given key-(record id) pair into the given leaf node.

#### Parameters

- origNode** a leaf node
- origPageId** the page id of the page that stores the leaf node
- key** the key of the key-record pair
- rid** the record id of the key-record pair
- midVal** a reference to an integer in the parent node. If the insertion requires a split in the leaf node, midVal is set to the smallest element of the newly created node.

#### Returns

The page number of the newly created page if insertion requires a split, or 0 if no new node is created.

### ◆ insertToNonLeafNode()

```

void badgerdb::BTreeIndex::insertToNonLeafNode ( NonLeafNodeInt * n,
                                                  int             i,
                                                  int             key,
                                                  PageId         pid
                                                  )

```

private

Inserts the given key-(page number) pair into the given leaf node at the given index.

#### Parameters

- n** an internal node
- i** the insertion index
- key** the key of the key-(page number) pair
- pid** the page number of the key-(page number) pair

### ◆ isLeaf()



```
bool badgerdb::BTreeIndex::isLeaf ( Page * page )
```

private

This method takes in a page and checks if the page stores a leaf node or an internal node.

Assumption: The level for leaf node is -1.

#### Parameters

**page** the page being checked

#### Returns

true if the page stores a leaf node false if the page stores an internal node

### ◆ isLeafNodeFull()

```
bool badgerdb::BTreeIndex::isLeafNodeFull ( LeafNodeInt * node )
```

private

Checks if a leaf node is full

Assumption: All valid records are nonzero.

#### Parameters

**node** a leaf node

#### Returns

true if a leaf node is full false if a leaf node is not full

### ◆ isNonLeafNodeFull()

```
bool badgerdb::BTreeIndex::isNonLeafNodeFull ( NonLeafNodeInt * node )
```

private

Checks if an internal node is full

Assumption: All valid page numbers are nonzero.

#### Parameters

**node** an internal node

#### Returns

true if an internal node is full false if an internal node is not full

### ◆ moveToNextPage()

```
void badgerdb::BTreeIndex::moveToNextPage ( LeafNodeInt * node )
```

private

Change the currently scanning page to the next page pointed to by the current page.

#### Parameters

**node** the node stored in the currently scanning page.

### ◆ scanNext()

```
const void badgerdb::BTreeIndex::scanNext ( RecordId & outRid )
```

Fetch the record id of the next index entry that matches the scan. Return the next record from current page being scanned. If current page has been scanned to its entirety, move on to the right sibling of current page, if any exists, to start scanning that page. Make sure to unpin any pages that are no longer required.

#### Parameters

**outRid** RecordId of next record found that satisfies the scan criteria returned in this

#### Exceptions

**ScanNotInitializedException** If no scan has been initialized.

**IndexScanCompletedException** If no more records, satisfying the scan criteria, are left to be scanned.

### ◆ setEntryIndexForScan()

```
void badgerdb::BTreeIndex::setEntryIndexForScan ( )
```

private

Find the first element in the currently scanning page that is within the given bound.

### ◆ setNextEntry()

```
void badgerdb::BTreeIndex::setNextEntry ( )
```

private

Continue scanning the next entry. If the currently scanning entry is the last element in this page, set the current scanning page to the next page.

### ◆ setPageIdForScan()

```
void badgerdb::BTreeIndex::setPageIdForScan ( )
```

private

Recursively find the page id of the first element larger than or equal to the lower bound given.

### ◆ splitLeafNode()

```
void badgerdb::BTreeIndex::splitLeafNode ( LeafNodeInt * node,  
                                           LeafNodeInt * newNode,  
                                           int           index  
                                           )
```

private

Splits a leaf node into two. It moves the records after the given index in the node into the new node.

#### Parameters

- node** a pointer to the original node
- newNode** a pointer to the new node
- index** the index where the split occurs.

### ◆ splitNonLeafNode()

```
void badgerdb::BTreeIndex::splitNonLeafNode ( NonLeafNodeInt * curr,  
                                              NonLeafNodeInt * next,  
                                              int               i,  
                                              bool              keepMidKey  
                                              )
```

private

Split the internal node by the given index. It moves the values stored in the given node after the split index into a new internal node.

#### Parameters

- node** an internal node
- i** the index where the split occurs.
- keepMidKey** Whether the value at the index should be moved to the parent internal node or not. If keepMidKey is true, then the pair at the index does not need to be moved up and will be moved to the newly created internal node.

#### Returns

a pointer to the newly created internal node.

## ◆ splitRoot()

```
PageId badgerdb::BTreeIndex::splitRoot ( int      midVal,  
                                           PageId pid1,  
                                           PageId pid2  
                                           )
```

private

Create a new root with midVal, pid1 and pid2.

### Parameters

**midVal** the first middle value of the new root

**pid1** the first page number in the new root

**pid2** the second page number in the new root

### Returns

the page id of the new root

## ◆ startScan()

```

const void badgerdb::BTreeIndex::startScan ( const void *    lowVal,
                                             const Operator lowOp,
                                             const void *    highVal,
                                             const Operator highOp
                                             )

```

Begin a filtered scan of the index. For instance, if the method is called using ("a",GT,"d",LTE) then we should seek all entries with a value greater than "a" and less than or equal to "d". If another scan is already executing, that needs to be ended here. Set up all the variables for scan. Start from root to find out the leaf page that contains the first RecordID that satisfies the scan parameters. Keep that page pinned in the buffer pool.

### Parameters

**lowVal** Low value of range, pointer to integer / double / char string

**lowOp** Low operator (GT/GTE)

**highVal** High value of range, pointer to integer / double / char string

**highOp** High operator (LT/LTE)

### Exceptions

**BadOpcodesException** If lowOp and highOp do not contain one of their their expected values

**BadScanrangeException** If lowVal > highval

**NoSuchKeyFoundException** If there is no key in the B+ tree that satisfies the scan criteria.

## Member Data Documentation

### ◆ attrByteOffset

```
int badgerdb::BTreeIndex::attrByteOffset {}
```

private

Offset of attribute, over which index is built, inside records.

### ◆ attributeType

```
Datatype badgerdb::BTreeIndex::attributeType
```

private

Datatype of attribute over which index is built.

### ◆ bufMgr

BufMgr\* badgerdb::BTreeIndex::bufMgr {}

private

Buffer Manager Instance.

### ◆ currentPageData

Page\* badgerdb::BTreeIndex::currentPageData {}

private

Current Page being scanned.

### ◆ currentPageNum

PageId badgerdb::BTreeIndex::currentPageNum {}

private

Page number of current page being scanned.

### ◆ file

File\* badgerdb::BTreeIndex::file {}

private

File object for the index file.

### ◆ highOp

**Operator** badgerdb::BTreeIndex::highOp {LT}

private

High Operator. Can only be LT(<) or LTE(<=).

### ◆ highValInt

int badgerdb::BTreeIndex::highValInt {}

private

High INTEGER value for scan.

#### ◆ lowOp

**Operator** badgerdb::BTreeIndex::lowOp {GT}

private

Low Operator. Can only be GT(>) or GTE(>=).

#### ◆ lowValInt

int badgerdb::BTreeIndex::lowValInt {}

private

Low INTEGER value for scan.

#### ◆ nextEntry

int badgerdb::BTreeIndex::nextEntry {}

private

Index of next entry to be scanned in current leaf being scanned.

#### ◆ scanExecuting

bool badgerdb::BTreeIndex::scanExecuting {}

private

True if an index scan has been started.

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

- src/**btree.h**