# CS387 Project Report

# Indexing in ToyDB

Anirudh Vemula (110050055)
Rajeev Kumar M (110050052)
Vamsidhar Yeddu (110050051)
Kartik Pranav K (110050062)

16th november 2013

## Parts of the report:

## Description of the project:

We sought to implement the uniquifier attribute approach in indexing and observe its performance against the bucket of pointers approach. We measure the performance using the time it takes to execute a query or modify the index and the memory it uses to store the index.

## Description of the framework:

We used the framework developed by TU Ilmenau named ToyDB. The ToyDB framework consists of two layers, the AM layer and the PF layer. We deal with the former, the Access Method layer, which is involved in maintaining secondary indices on the attributes of the database.

ToyDB originally implements a bucket of pointers approach to store the indices as a B+ tree. We modify this design to support the uniquifier attribute approach. ToyDB manages the

buffer memory and file/page handling using the PF layer, which we have left unexplored and untouched.
The B+ tree nodes are stored as pages of a file using the PF layer as a black box. The AM layer derives some abstract functions from the PF layer to achieve this. The whole framework is written in C and implements separate functions for different utilities such as search, insert, delete and scan.

## Code Structure:

The code of ToyDB is divided into two folders, aptly named ‘amlayer’ and ‘pflayer’. Our modifications in the code exist completely in the AM layer. We have significantly modified the search, insert, delete and scan functions to suit our approach.

The test cases used in the project exist in a separate folder named ‘testcases’. The test cases cover both the checks for correctness and performance comparisons. Makefiles have been provided to compile them and observe their outputs.

The complete (descriptive) documentation of the framework and our modifications has been recorded in three parts:

1. LaTeX document attached as an appendix to this report
2. Doxygen documentation in the code folder (in /doc/)
3. Extensive comments in the code

## Uniquifier attribute approach:

The exact details of our approach and the design used in the project can be found in the slides used in the demonstration that are attached as an appendix to this report. Any other minor details can be found in the documentation.

## Test Plan:

**Control Parameters**

1. Number of values inserted
2. Repeating/distinct attribute values
3. Attribute type
4. Operation used in the query (applies only to scan)
5. The query/modification itself

**Measurement Parameters**

1. The execution time to execute a query or a modification to the database (insert/delete)
2. The memory used, which is measured by the number of B+ tree nodes created (or pages, in our case)

We conduct the following experiments BEFORE and AFTER our project modification and report the results in the next section.

**Experiments**

1. Execution time and the memory used for inserting a large number of records.
2. Execution time for deleting a large number of records and the number of pages left.
3. Execution time for scanning an index using different operations on different attribute types.

**Test cases**

We have included several test cases for both the correctness and performance measurement in our submission for the project.

## Results of the experiments conducted:

### Experiment 1:

Inserting a large number of records with distinct attribute values (integer type)

**Approach 1 (uniquifier):**

time : 0.0155 seconds

Number of pages used : 181

**Approach 2 (bucket of pointers):**

time : 0.0052 seconds

Number of pages used : 126

### Experiment 2:

Inserting a large number of records with repeating attribute values (character type)

**Approach 1:**

time : 0.0062 seconds

Number of pages used : 17

**Approach 2:**

time : 0.0009 seconds

Number of pages used : 19

### Experiment 3:

Deleting a large number of records with distinct attribute values (character type)

**Approach 1:**

time : 0.012 seconds

Number of pages left : 122

**Approach 2:**

    time : 0.0055 seconds
    Number of pages left : 122

__**Experiment 4:**__
Deleting a large number of records with repeating attribute values (character type)
**Approach 1:**
    time : 0.0036 seconds
    Number of pages left : 25
**Approach 2:**
    time : 0.0015 seconds
    Number of pages left : 25

__**Experiment 5:**__
Scanning through a large number of records with repeating attribute values (character type) using the equality operator.
**Approach 1:**
    time : 0.027 seconds
**Approach 2:**
    time : 0.003 seconds

__**Experiment 6:**__
Scanning through a large number of records with repeating attribute values (character type) using the greater than operator.
**Approach 1:**
    time : 0.027 seconds
**Approach 2:**
    time : 0.008 seconds

__**Experiment 7:**__
Scanning through a large number of records with repeating attribute values (integer type) using the less than or equal

to operator.
**Approach 1:**
    time : 0.028 seconds
**Approach 2:**
    time : 0.002 seconds

All the correctness checks have also been included in the folder. Each test case checks the correctness of exactly one function (each operation in separate file, in the case of scan).

## Conclusions:

1. The bucket of pointers approach proves to be more memory-efficient than the uniquifier attribute approach as can be seen in the number of pages (nodes) used in the test cases.
2. The uniquifier approach also loses out to the bucket of pointers in terms of time taken to execute a query/modify a database. This is proved by the performance measurements done in the test cases.
3. In effect, the bucket of pointers is a very viable and efficient approach to indexing when compared to uniquifier attributes, although its implementation is rather complicated. The latter is suggested only if simplicity of implementation is required as part of the development scheme.

### What went well

We have grasped how one should go about understanding a code base as huge as ToyDB. We have gained a lot of experience in C coding through this project and our fundamentals were revised. We have understood the inner workings of a database engine when it comes to indexing the attributes and how B+ trees function.

**What didn’t go well**

Due to the lack of documentation, we had to spend several days understanding the code which we didn’t split amongst ourselves. This resulted in two guys understanding the code and then explaining it to the other two guys and then we together started coding. If we would have distributed our work of understanding the code, the project would have gone more smoothly.

# Documentation of the AM Layer of ToyDB

Anirudh Vemula   Rajeev Kumar M
Vamsidhar Yeddu   Kartik Pranav K

November 16, 2013

# Access Method (AM) Layer

AM layer is an abstracted layer that helps to support an indexed file abstraction, where files are organised as heaps and may have any number of single-attribute secondary indices associated with them to speed up selection and equality-join queries. In this layer, we borrow the utility of Paged File (PF) layer to represent a B+ tree index, and we use pages in the file to represent individual tree nodes.

AM layer supports *inserting*, *deleting* and *scanning* on a B+ tree index. There are separate routines to implement these functionalities in addition to interfaces provided for creating and destroying an index. Scanning in AM layer supports operations such as: equality, greater than, less than, greater than or equal to, less than or equal to and not equal.

AM layer only supports indexing on single attribute and doesn't support multi-attribute indexing. The indexing employed by AM layer indexes on the attribute and stores pointers to memory (on the same page) containing record IDs rather than the actual records. The underlying memory manipulations and the file page handling is done by the PF layer.

## 1 Major routines in AM Layer

This section describes all the major routines that a user can use to create indices, insert entries, delete entries and scan through the entries.

### 1.1 `AM_CreateIndex(fileName, indexNo, attrType, attrLength)`

#### 1.1.1 Arguments

- `fileName` : name of the indexed file
- `indexNo` : index number for this file
- `attrType` : type of the indexed attribute
- `attrLength`: Length of the indexed attribute in bytes

#### 1.1.2 Return Value

- `AME_INVALIDATTRLENGTH` : If the given attrLength is invalid

- `AME_INVALIDATTRTYPE` : If the given attrType is invalid
- `AME_PF` : If there is an error in opening/handling the file
- `AME_OK` : If the index is successfully created

#### 1.1.3 Description

This function, as the name suggests, creates an index in a file given by `fileName` with index number `indexNo`. This function creates a file named `fileName.indexNo` to store the index in. It then allocates memory and creates a new page in the file in which it stores the root node. This root node is initially a leaf node as there are no keys yet, so the page header is initialised as a leaf header and the appropriate parameters are initialised. We write this header onto the page, then close the file and set the `AM_RootPageNum` parameter to the page number of the newly created page.

### 1.2 `AM_DestroyIndex(fileName, indexNo)`

#### 1.2.1 Arguments

- `fileName` : name of the indexed file
- `indexNo` : index number for this file

#### 1.2.2 Return Value

- `AME_PF` : If there is an error in opening/handling the file
- `AME_OK` : If the index is successfully destroyed

#### 1.2.3 Description

This function destroys the index specified by the `indexNo` stored in the file `fileName`. Essentially, it destroys the file named `fileName.indexNo`.

### 1.3 `AM_InsertEntry(fileDesc, attrType, attrLength, value, recID)`

#### 1.3.1 Arguments

- `fileDesc` : File descriptor
- `value` : value to be inserted into the index

- `redid` : ID of the record whose attribute value is being inserted

### 1.3.2 Return value

- `AME_INVALIDATTRTYPE`
- `AME_INVALIDATTRLENGTH`
- `AME_INVALIDVALUE` : If value is NULL
- `AME_FD` : If the file descriptor is less than zero i.e. invalid file descriptor
- return value less than zero implies an error in either the search or insert or split
- `AME_OK` : Insertion was successful

### 1.3.3 Description

This function initially searches the index for the given value to check whether it already exists or not. If exists, the page number of its location and if not exists, the page number of its expected location if it was inserted, is stored. This information is passed onto the `AM_InsertintoLeaf` function to actually insert the (value, recID) pair into the index. This function returns true if it was successfully inserted. If it returns false, then the leaf was full and needs to be split which is accomplished by the `AM_SplitLeaf` function. This function returns true, if there is a value that needs to be inserted into the parent due to the split. If true, then the `AM_AddtoParent` function is called. If at any stage of this whole process, a function fails then a negative value is returned by `AM_InsertEntry` function.

## 1.4 `AM_DeleteEntry(fileDesc, attrType, attrLength, value, recID)`

### 1.4.1 Arguments

- `fileDesc`
- `attrType`
- `attrLength`
- `value` : value to be deleted
- `recID` : record ID of the record whose attribute value is being deleted

### 1.4.2 Return value

- `AME_FD`
- `AME_INVALIDATTRTYPE`
- `AME_INVALIDATTRLENGTH`
- `AME_INVALIDVALUE` : If value is NULL
- `AME_NOTFOUND` : If the given value is not found in the index
- `AME_OK` : If the deletion was successful

### 1.4.3 Description

This function initially searches for the page and index at which the value exists. If the value is not found, it returns `AME_NOTFOUND`. If found, then it follows the pointer stored at the index to the list of recIDs corresponding to records with the same value. It then iterates through this list to find the appropriate recID and deletes it. It then updates the free list (stored in header) accordingly. In the case where the record currently being deleted is the last record for the given value, it deletes the key too and displaces all the keys present after it by one place.

## 1.5 `AM_OpenIndexScan(fileDesc, attrType, attrLength, op, value)`

### 1.5.1 Arguments

- `fileDesc`
- `attrType`
- `attrLength`
- `op` : The operation used for comparison
- `value` : value on which the comparison is made

### 1.5.2 Return value

- `AME_FD`
- `AME_INVALIDATTRTYPE`
- `AME_SCAN_TAB_FULL` : If you have exceeded the maximum number of scans
- `AME_PF`
- return value less than zero implies an error

### 1.5.3 Description

This function sets up the framework to scan through the index according to the user's preferences. If `value` is NULL, then irrespective of the operation, the scan iterates through all the records in the index. If the `value` is not NULL, then the scan table is appropriately set to the next index and page. The bounds for the scan are also set according to the operation listed in `op`. After setting up the scan table, a scan descriptor is returned.

## 1.6 `AM_FindNextEntry(scanDesc)`

### 1.6.1 Arguments

- `scanDesc` : The scan descriptor

### 1.6.2 Return Value

- `AME_INVALID_SCANDESC` : The scanDesc is invalid
- `AME_EOF` : The scan has reached its end
- record ID of the next entry in the scan

### 1.6.3 Description

This function uses the framework set up by `AM_OpenIndexScan` to iterate through all the entries satisfying the scan criterion. This function makes the appropriate boundary checks and terminates the scan when there is no more to scan. Each call of this function returns the next ID of the record which satisfies the scan criterion.

### 1.7 `AM_CloseIndexScan(scanDesc)`

#### 1.7.1 Arguments

- `scanDesc` : The scan descriptor

#### 1.7.2 Return Value

- `AME_INVALID_SCANDESC` : The scanDesc is invalid
- `AME_OK`

#### 1.7.3 Description

This function takes the scan descriptor and terminates the scan corresponding to it.

# 2 Other Routines in AM Layer

## 2.1 `AM_InsertintoLeaf(pageBuf, attrLength, value, recId, index, status)`

### 2.1.1 Arguments

- `pageBuf` : Buffer where the leaf page resides
- `attrLength`
- `value` : The value that you intend to insert
- `recId`
- `index` : index where the key is to be inserted
- `status` : Whether the key is a new key or an old one

### 2.1.2 Return Value

- TRUE : If the insertion was successful
- FALSE : If the insertion failed due to lack of room

### 2.1.3 Description

If the status is `AM_FOUND`, then the key is an old one. So a record must be inserted. For this, the function first checks whether the freeList is empty or not. If empty, then it checks whether there is enough space for a new record. If not, it returns FALSE. If there is space, it calls the `AM_InsertToLeafFound` function to insert the recId. If the freeList is not empty and there is not enough space to insert a record, then `AM_Compact` is called to create space for a record by compacting the free list. If status is `AM_NOTFOUND`, then the key is a new key. The same check of checking the space remaining is done. If there is non, then it returns FALSE. If there is enough space, then `AM_InsertToLeafNotFound` function is called to insert the value and recID.

## 2.2 `AM_InsertToLeafFound(pageBuf, recID, index, header)`

### 2.2.1 Arguments

- `pageBuf`
- `recID`
- `index`
- `header`

### 2.2.2 Return Value

No return Value.

### 2.2.3 Description

If the free list is not empty, then the function inserts the recID into the first slot of the free list. If it is empty, then the function inserts the recID into the list of recIDs list at the head of it. It makes appropriate change of pointers so that the head of the recID list now points to the newly added recID and other such important changes.

## 2.3 `AM_InsertIntoLeafNotFound(pageBuf, value, recID, index, header)`

### 2.3.1 Arguments

- `pageBuf`

- `value`
- `recID`
- `index`
- `header`

### 2.3.2 Return Value

No return value.

### 2.3.3 Description

This function creates the space for the new key by pushing all the keys greater than this to the right. It then updates the KeyPtr in the header to increment by an amount of one record size. the new value is copied into the space created and then the `AM_InsertIntoLeafFound` function is called, as if the inserted key was an old one.

## 2.4 `AM_Compact(low, high, pageBuf, tempPage, header)`

### 2.4.1 Arguments

- `low` : lower index
- `high` : higher index until which compact needs to be done
- `pageBuf`
- `tempPage` : Temporary page created after compacting pageBuf from low to high
- `header` : header of pageBuf

### 2.4.2 Return value

Returns no value but modifies the tempPage given as input.

### 2.4.3 Description

This function copies all the keys from low to high and their recIDs to a temporary page `tempPage`. We can use this function to compact the free list and create some space for insertion of a new key/record by compacting the original page and writing into the same page.

## 2.5 AM_Search(fileDesc, attrType, attrLength, value, pageNum, pageBuf, indexPtr)

### 2.5.1 Arguments

- `fileDesc` : file descriptor
- `attrtType`
- `attrLength`
- `value` : The value that is searched for
- `pageNum` : page number of the page where key is present or can be inserted
- `pageBuf` : pointer to buffer in memory where the leaf page corresponding to pageNum can be found
- `indexPtr` : pointer to index in leaf where key is present or can be inserted

### 2.5.2 Return value

- `AME_INVALIDATTRTYPE`
- `AME_INVALIDATTRLENGTH`
- `AM_NOTFOUND` : If not found
- `AM_FOUND` : If found

### 2.5.3 Description

This function searches the file given by `fileDesc` for the page where the `value` is present or can be inserted. Upon finish, it returns the page number, index and brings the concerned page into the memory pointed by `pageBuf`. It accomplishes this by doing a binary search of the value in the internal nodes(starting from root) and progressing downward the B+ tree accordingly. Upon reaching a leaf, it calls the `AM_SearchLeaf` function which returns the index for the value in the leaf page.

## 2.6 `AM_BinSearch(pageBuf, attrType, attrLength, value, indexPtr, header)`

### 2.6.1 Arguments

- `pageBuf` : Buffer where the page is found
- `attrType`
- `attrLength`
- `value`
- `indexPtr` : index in the page at which the value is found
- `header` : header of the page passed as a parameter by `AM_Search`

### 2.6.2 Return Value

Returns the page number of the subsequent page to go to in `AM_Search`.

### 2.6.3 Description

This functions implements the binary search used by ToyDB to find the subsequent page to go to while searching for a value in a B+ tree. It is used to achieve binary search on the keys stored in an internal node and get the page number of the next page accordingly. It uses the `AM_Compare` function to compare the keys in the node with the given value. Upon finish, it returns the page number of the subsequent page the search should proceed on and the indexPtr at which this page number is found on the internal node.

## 2.7 `AM_SearchLeaf(pageBuf, attrType, attrLength, value, indexPtr, header)`

### 2.7.1 Arguments

- `pageBuf`
- `attrType`
- `attrLength`
- `value`
- `indexPtr`
- `header`

#### 2.7.2 Return Value

- `AM_NOTFOUND`
- `AM_FOUND`

#### 2.7.3 Description

This function searches the leaf page given (pointed by `pageBuf`) for the value and returns whether it actually found the value or not in the leaf. It employs binary search to accomplish this. If found, it points the `indexPtr` to the index where it found the value. If it doesn't find the value, then it points the `indexPtr` to where the value should be inserted.

### 2.8 `AM_Compare(bufPtr, attrType, attrLength, valPtr)`

#### 2.8.1 Arguments

- `bufPtr` : Points to one of the value to be compared (in buffer)
- `attrType`
- `attrLength`
- `valPtr` : Points to the other value to be compared

#### 2.8.2 Return value

This function returns:

- 0 : If the values compared are identical
- 1 : If the value pointed by `valPtr` is larger than `bufPtr`
- -1 : If the value pointed by `valPtr` is smaller than `bufPtr`

#### 2.8.3 Description

This function compares the value pointed by `valPtr` with the value pointed by the `bufPtr`. It first checks the `attrType` and then uses the appropriate comparison function to compare them.

## 2.9 `AM_SplitLeaf(fileDesc, pageBuf, pageNum, attrLength, recId, value, status, index, key)`

### 2.9.1 Arguments

- `fileDesc`
- `pageBuf`
- `pageNum` : Page number of the new leaf created is stored in this
- `attrLength`
- `recID`
- `value` : Value to be inserted (that caused this split)
- `status` : Whether the key(value) is present in the tree or not
- `index` : place where the key is to be inserted
- `key` : returns the key to be filled in the parent, if there is any

### 2.9.2 Return Value

- FALSE : If there is no key to be filled in the parent
- TRUE : If there is a key to be filled in the parent

### 2.9.3 Description

This function splits the leaf when the leaf becomes full and there is no more space to insert a new key. It first compacts (using `AM_Compact`) the first half of its keys(and their recIDs) into a temporary page and then allocates a new page for the remaining half. If the split leaf is also the root of the page, then a new page is allocated and is filled with a header with root parameters and the first key of the second leaf is added to it as the first key. If not, then the value to be added to the parent is stored in `key` and it returns TRUE.

## 2.10 `AM_AddtoParent(fileDesc, pageNum, value, attrLength)`

## 2.11 Arguments

- `fileDesc`
- `pageNum` : page number to be added to the parent

- `value` : pointer to the attribute value to be added to the parent
- `attrLength`

#### 2.11.1 Return Value

- `AME_PF` : If any error arises in page handling
- `AME_OK` : If the insertion was successful

#### 2.11.2 Description

This function adds to the parent(an internal node) the first key of the new page created due to the split. It first checks if there is enough space for a new key in the internal node. If there is, then it directly inserts the key and the associated page number(of the new page). If there isn't, then it splits the internal node using `AM_SplitIntNode` function. If the split internal node isn't root, then it recursively calls `AM_AddtoParent` function. Else, it creates a new page for the root and fills it with the appropriate header.

### 2.12 `AM_SplitIntNode(pageBuf, pbuf1, pbuf2, header, value, pageNum, offset)`

#### 2.12.1 Arguments

- `pageBuf` : Buffer containing the internal node to be split
- `pbuf1` : buffer for one half
- `pbuf2` : buffer for the other half
- `header` : header of the internal node to be split
- `value` : pointer to the key to be added
- `pageNum`
- `offset`

#### 2.12.2 Return value

Returns no value.

### 2.12.3 Description

This function splits the internal node in the same way `AM_SplitLeaf` splits the leaf node.

## 3 Macros in AM Layer

- `AM_NOT_FOUND` : 0
- `AM_FOUND` : 1
- `AM_NULL` : 0
- `AM_NULL_PAGE` : -1
- `AME_OK` : 0
- `AME_INVALIDATTRLENGTH` : -1
- `AME_NOTFOUND` : -2
- `AME_PF` : -3
- `AME_INTERROR` : -4
- `AME_INVALID_SCANDESC` : -5
- `AME_INVALID_OP_TO_SCAN` : -6
- `AME_EOF` : -7
- `AME_SCAN_TABLE_FULL` : -8
- `AME_INVALIDATTRTYPE` : -9
- `AME_FD` : -10
- `AME_INVALIDVALUE` : -11
- `AM_si` : sizeof(int)
- `AM_ss` : sizeof(short)
- `AM_sl` : sizeof(leaf header)
- `AM_sint` : sizeof(internal header)

- `AM_sc` : sizeof(char)
- `AM_sf` : sizeof(float)

# Indexing Optimization in ToyDB

CS 387 Final Project Demo
12-11-2013

Team:
Anirudh Vemula
Rajeev Kumar M
Vamsidhar Y
Kartik Pranav K

## Aim of the project

- Using AM layer, create a secondary index on a repeating (non-unique) attribute of a file (whose records are not in the key order) by making the keys unique by appending the primary key(or record ID) i.e. implementing the uniquifier attribute approach. Study the space and performance characteristics of the uniquifier attribute implementation when compared to the implementation done by ToyDB.

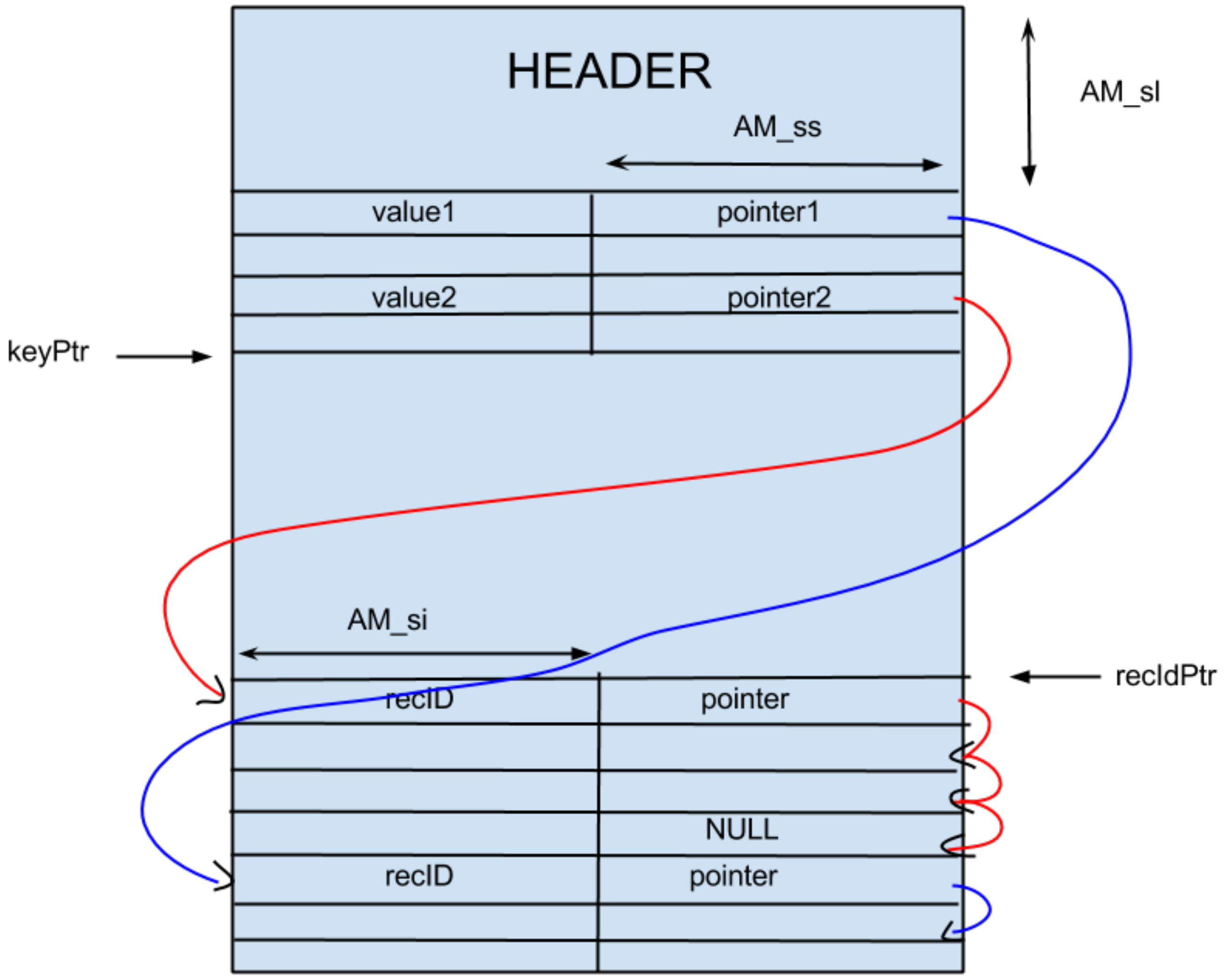
## Project Introduction

- We have taken ToyDB, a fully working DBMS developed by TU Ilmenau.
- We are optimising the indexing for attributes with repeating values by implementing the uniquifier attribute approach.
- After the implementation, we compare its performance with the previous ToyDB implementation.

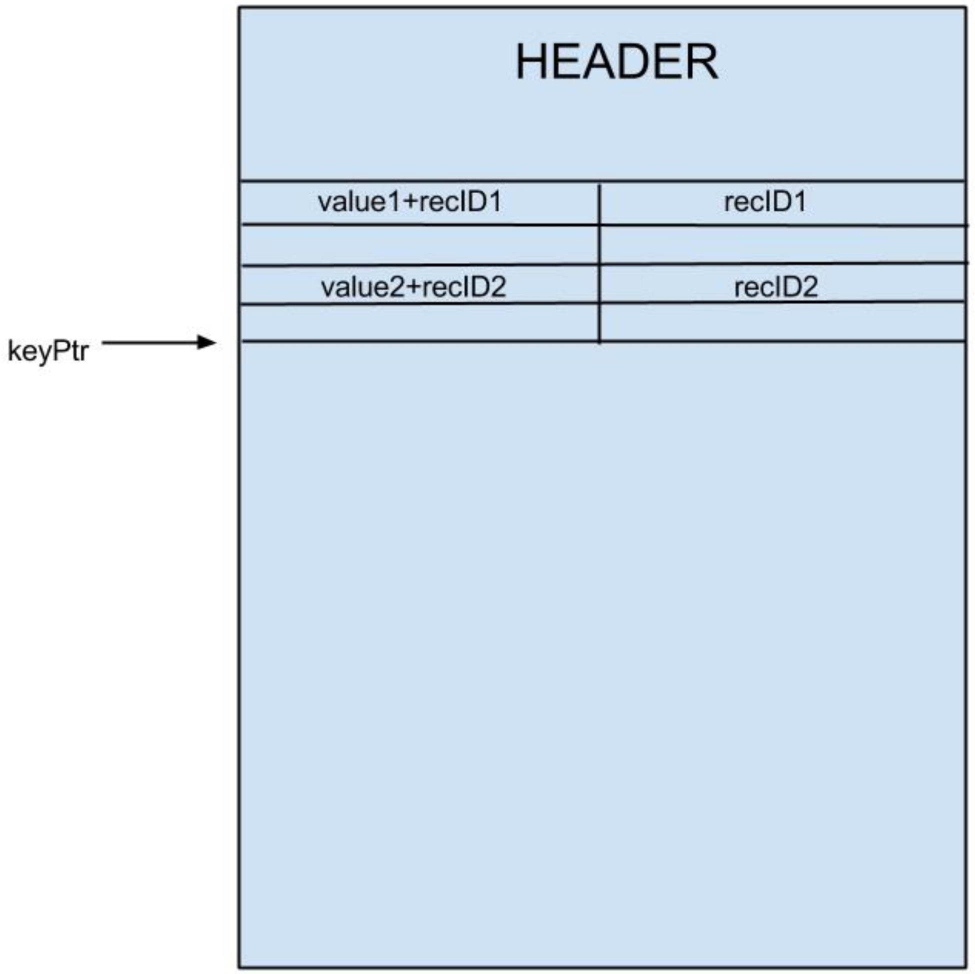
## Design Details

- For implementing uniquifier attribute approach, we significantly changed many functions used for search, insert, delete and scan.
- We changed the leaf page format and the key format so that it suits our implementation.
- We have included several identifiers so that indexing is done efficiently and in a simple way.

# ToyDB’s Leaf Page Format



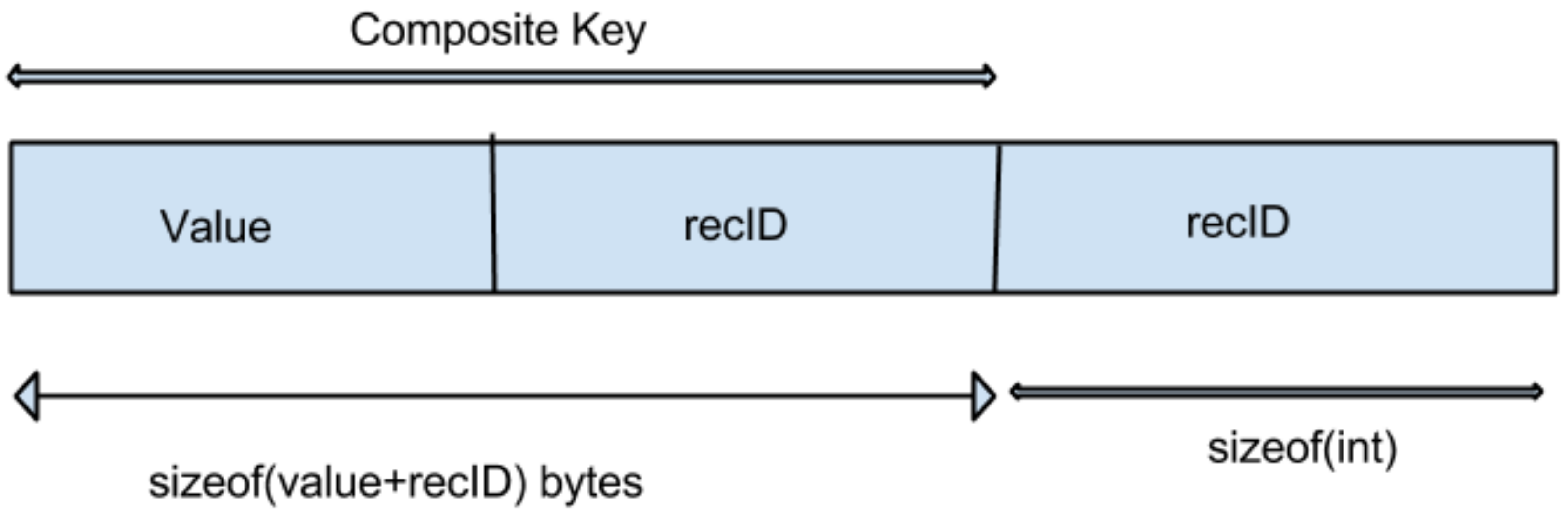
# Modified Leaf Page Format



## Modified Leaf Page Format

- We have removed the recID pointers list that was maintained earlier. This is because each key value (value + recID) in our case has a unique recID associated with it.
- In the place of pointers, we are explicitly storing the recID along with the key value.
- The reason for the above is explained in the following slides.

## How keys are stored on pages



## How keys are stored on pages

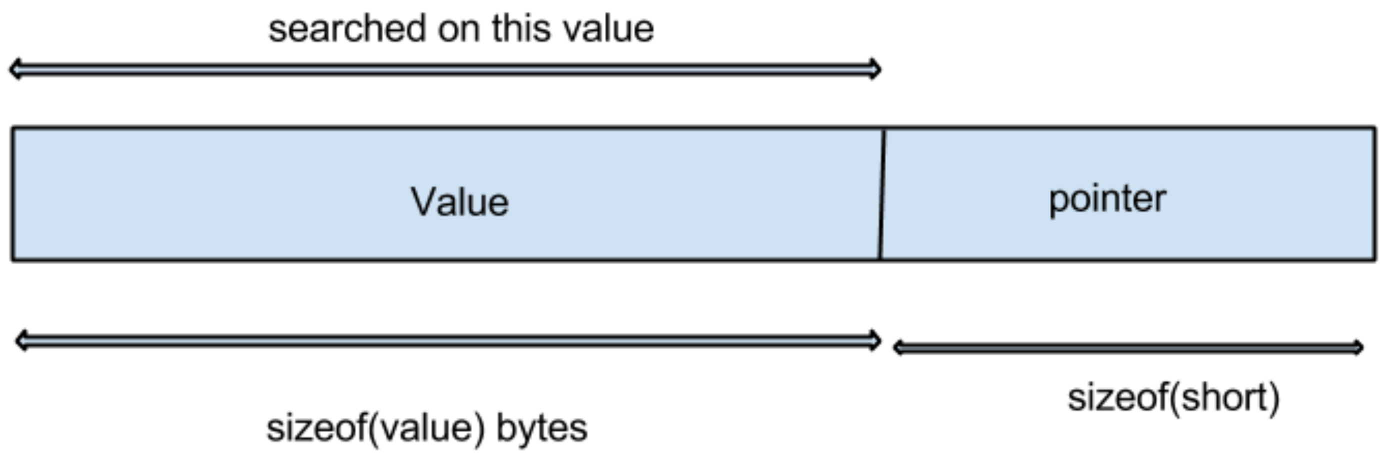
- The value on which the index is searched on is the composite key depicted above.
- We also store the recID as a separate entity in the record.
- In the following slides, we show how we have exploited the value of recID stored in the composite key to improve/simplify indexing on the attribute.

## Header

- We haven’t made any significant changes to the header used by ToyDB initially except to delete some redundant features such as freelist and maintaining recID pointers.
- The free list is maintained by the previous implementation to identify deleted records, but in our case we don’t explicitly store the records. So, there is no use of free list in our implementation.

## Search

- The Key format used by ToyDB is depicted below. In this case, all the search has to do is to search for the right value and then follow the pointer to get all the records with same key.



## Search

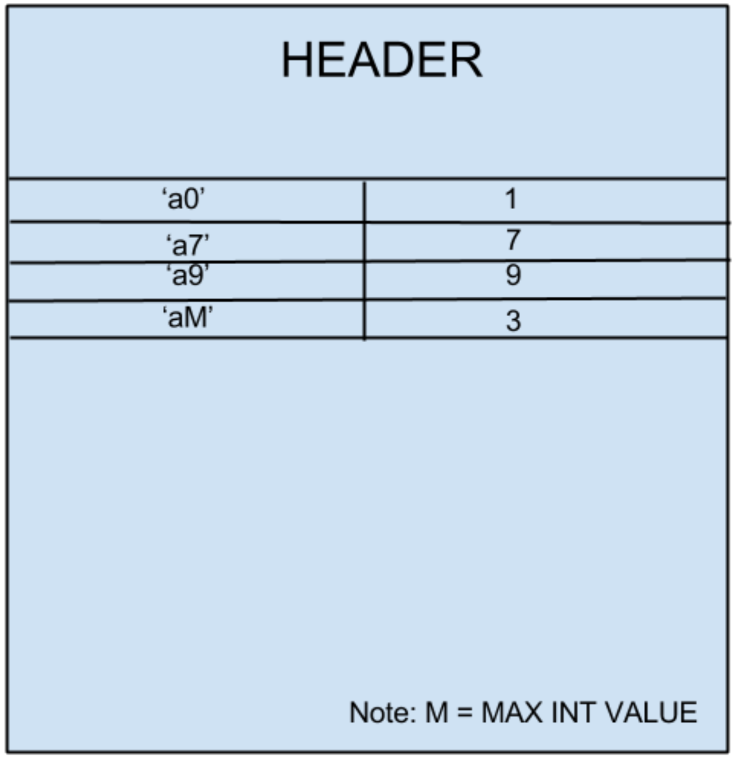
- But in our implementation, we need the (value+recID) to search. In the insert and delete functions where we have access to recID these can be implemented straightforwardly.
- However, in the case of scan we don’t have recID. In this case, to search appropriately, we need adequate identifiers.

## Insert

- We resolve the issue of unavailability of recID by ensuring that the first key inserted for a given value has the appended recID value zero and the second key inserted to have the appended recID value MAX_INT.
- This makes sure that any record inserted for the same value has key that lies between the above two keys.
- And for any scan, we can uniquely identify the start and end keys corresponding to the same value.

## Insert Example

- We insert the following (value,recID) pairs in the given order: ('a',1),('a',3),('a',9),('a',7)
- The leaf page resulting looks like:



## Insert Procedure

During insertion of a new key for a value(say 'a')

1. We search the tree for the value 'a' with the searched value length = sizeof('a') (instead of sizeof('a')+sizeof(int))
2. If not found, we search for the existence of 'a0'. If not found, we insert ('a0',recID) into the tree.
3. If found, we search for the existence of 'aM'. If not found, we insert ('aM',recID) into the tree.
4. If found, we just insert ('arecID',recID) into the tree.

## Deletion

- Deletion is similar to insertion. We conduct the same checks as before to ensure that there are always identifiers for any value.
- In the case where an identifier for a value is deleted, we make sure that the next key in the page with the same value becomes the identifier.

## Scanning

- In scanning, our approach of uniquifier attribute fails with the previous implementation of the operations EQUAL, LESS THAN EQUAL, GREATER and NOT EQUAL.
- We had to significantly modify these operations so that they take into account the fact that multiple records with same value do not exist as a single key rather as multiple keys

## Scanning

- We use the identifiers that we have placed while inserting to identify the start and the end of keys corresponding to the same value for the attribute.

## Scanning procedure

If we scan the previous example to get all records with value equal to 'a', we

1. We first search for the page and index at which 'a0' is present and store the index and page
2. We then search for the page and index at which 'aM' is present and store the index and page (If it exists)
3. We now iterate through the pages between the above two pages, restricting to indices greater than or equal to index found on the first page and indices less than or equal to the index found on the last page.

## Scanning

- Although, the above procedure is specific to the EQUAL operation, we have similar procedures for the other operations too.
- In cases where there is only a single key corresponding to a value, we make the appropriate checks and modifications.

## Demo

- *We have implemented the project according to the above design and its correctness is tested.*
- *However, we haven't yet tested the performance of both the approaches on repeating attribute test cases. These performance measurements will be provided in the final project report.*

## Difficulties Faced

- We would like to highlight the lack of any kind of documentation for ToyDB. Due to this, we had to spend the first 3-4 days (in the implementation window) just to go through each line of code and try to understand.
- The fact that the whole project was based on pointers and their manipulations resulted in the above understanding being extremely difficult.
- We hope that the documentation of our project serves to be useful for any other person who wants to take up ToyDB in his project.

## Conclusion

- We have implemented the uniquifier attribute approach to indexing.
- We will compare the performance of this approach with the original approach employed by ToyDB

## Acknowledgements

- We would like to thank our seniors Piyush and Umang for their help regarding PF layer and debugging some of PF Layer errors.

## Thank You – Questions?