Database Internals Project Report

Katuri Sai Kiran  Ravichandra  Kranthi
130050051      130050061      130050062

November 6, 2015
1 Introduction

B+ trees are extensively used in Database Management Systems because search operation is much faster in them compared to B-trees. This is primarily because unlike B-trees, B+trees have very high fan out, which reduces the number of I/O operations required to find an element in the tree. This makes the insertion, deletion and search using B+ trees very efficient.

However the downside of having just the operations of insertion, deletion and search, is that insertion of large collections of data records is quite expensive, because each entry requires us to start from the root and go down to the appropriate leaf page. Hence we would also like to implement the efficient alternative, bulk-loading.

2 Motivation

Consider the case where a B+ tree is being built on a large relation. Suppose the relation is significantly larger than main memory, and we are constructing a non-clustering index on the relation such that the index is also larger than main memory. In this case, as we scan the relation and add entries to the B+ tree, it is quite likely that each leaf node accessed is not in the database buffer when it is accessed, since there is no particular ordering of the entries. With such randomly ordered accesses to blocks, each time an entry is added to the leaf, a disk seek will be required to fetch the block containing the leaf node. The block will probably be evicted from the disk buffer before another entry is added to the block, leading to another disk seek to write the block back to disk. Thus a random read and a random write operation may be required for each entry inserted.

For example, if the relation has 100 million records, and each I/O operation takes about 10 milliseconds, it would take at least 1 million seconds to build the index, counting only the cost of reading leaf nodes, not even counting the cost of writing the updated nodes back to disk. This is clearly a very large amount of time. In contrast, if each record occupies 100 bytes, and the disk subsystem can transfer data at 50 megabytes per second, it would take just 200 seconds to read the entire relation.

An efficient solution to this is to perform bulk loading of an index as follows. First, create a temporary file containing index entries for the relation, then sort the file on the search key of the index being constructed, and finally scan the sorted file and insert the entries into the index. There are efficient algorithms for sorting large relations.
There is a significant benefit to sorting the entries before inserting them into the B+ tree. When the entries are inserted in sorted order, all entries that go to a particular leaf node will appear consecutively, and the leaf needs to be written out only once. nodes will never have to be read from disk during bulk load, if the B+ tree was empty to start with. Each leaf node will thus incur only one I/O operation even though many entries may be inserted into the node.

If each leaf contains 100 entries, the leaf level will contain 1 million nodes, resulting in only 1 million I/O operations for creating the leaf level. Even these I/O operations can be expected to be sequential, if successive leaf nodes are allocated on successive disk blocks, and few disk seeks would be required. With current disks, 1 millisecond per block is a reasonable estimate for mostly sequential I/O operations, in contrast to 10 milliseconds per block for random I/O operations.

3 Problem Description and objectives

Given a collection of data records, we want to create a B+ tree index on some key field. One approach is to insert each record into an empty tree. However, it is quite expensive, because each entry requires us to start from the root and go down to the appropriate leaf page. An efficient alternative is to use bulk-loading. Hence, we want to implement bulk loading and compare its performance with normal B+ tree insertion, with different inputs and document the results.

4 Algorithm

4.1 Algorithm Description

1. Sort the data according to search keys in increasing order
2. Create an empty B+ tree root node
3. Add the entries in the sorted order to the root node till it is full
4. Create a new root node, and a new leaf node, and add a pointer in the root node to the two leaf nodes containing the indexes.
5. Keep inserting (search key, value) pairs in the sorted order in the right-most leaf node till it is full.
6. If the right most leaf node is full, create a new leaf node and the index entry into it.
7. Also add a pointer pointing to the newly created leaf node to the rightmost node one level above the leaf.

8. If the right most parent node is also full, then create a new non-leaf node and add a pointer to the newly created node to its parent.

9. Keep doing this iteratively to the top. If this proceeds till the root, then we have to split the root node, and create a new root.

Note that in the above algorithm description, we only add index, value entries to the right most node at any level. This works only because we have the entries to be inserted in sorted order.

4.2 Pseudocode

Algorithm 1 Bulk Loading in B+ Tree

1: char* rightMostBuffer[height] //stores the rightmost node at each level
2: procedure INSERTLEAF(value, key)
3:     if rightMostBuffer[0] = full then
4:         create newLeafNode
5:         newLeafNode.add(value, key)
6:         parentNode.add(value, key)
7:         return false
8:     else
9:         rightMostBuffer[0].add(value, key)
10:     return true
11:
12: procedure ADDTOPARENT(level, key)
13:     if rightMostBuffer[level] = full then
14:         create newInternalNode
15:         newInternalNode.add(value, key)
16:         AddtoParent(level+1, key)
17:     else
18:         rightMostBuffer[level].add(value, key)
19:
20: procedure ADDENTRYTOINDEX(value, key)
21:     status = InsertLeaf(value, key)
22:     if status is false then
23:         AddtoParent(0, key)
5 Implementation details

- Since at any time we need access only to the right-most nodes at all levels of the B+ tree, we fix only these nodes into memory.
- Whenever a rightmost node is split and a new node is created we no longer need to modify the entries in it. So, we unfix that page from the memory and write it back to the disk.
- We use functions similar to those described in normal insertion implementation provided in am layer(aminsert.c).
- A function named InsertLeaf checks if it is possible to insert a search key, value pair in the rightmost leaf node. If it is possible, it inserts. Else, we create a new leaf node and make a call to AddtoParent function with appropriate arguments.
- AddtoParent method inserts the value, pointer entry into the rightmost node on the level which is passed as an argument. If this results in a split, it recursively calls on its parent till the root.

6 Input, Outputs and Data Structures

1. We are planning to compare the bulk loading with normal insertion by using the following statistics.
   - Number of pages brought from disk to memory
   - Number of pages written to disk
   - Total time for creating the index table.

2. Thus we will take relations with different cardinality as inputs and as output, we will produce all the above mentioned statistics using normal insertion vs bulk loading for different cardinality of relations.

3. We will then compare the statistics and document the importance of bulk loading by drawing various plots for the relevant statistics.

4. We will use some data structures present in toyDB code for this purpose. We will use the AM layers data structures like am_leafheader which is a struct for leaf header and am_intheader which is a header for internal nodes.
5. We will use all the PF layer data structures but mostly we will not modify them unless there is some need to modify in-order to calculate some statistics.

6. We are planning to write all bulk loading related code in a separate file and use the stated data structures from toyDB inside the functions.

7 Brief Overview of Code

Our code is organized into three files similar to the default implementation given in the anlayer of toydb.

- *InsertEntry.c* has the following methods:

```c
/* Inserts a value,recId pair into the tree */
InsertEntry(fileDesc,attrType,attrLength,value,recId,last)
int fileDesc; /* file Descriptor */
char attrType; /* 'i' or 'c' or 'f' */
int attrLength; /* 4 for 'i' or 'f', 1-255 for 'c' */
char *value; /* value to be inserted */
int recId; /* recId to be inserted */
int last; /* Says whether the inserted entry is last one to be inserted while bulk loading*/
```

The above function is called in main function to insert an entry into the B+ tree. Therefore this function has to make all the necessary calls to insert leaf, parent node, split appropriate nodes etc..

```c
/* This function splits the leaf node and creates a new leaf and adds the new entry to the new leaf. Returns false on errors */
SplitLeaf(fileDesc,pageNum,attrType,attrLength,recId,value,last)
int fileDesc; /* file descriptor */
// char *pageBuf; /* pointer to buffer */
int pageNum; /* pageneumber of new leaf created */
char attrType;
```
int attrLength;
int recId;
char *value; /* attribute value for insert */
// char *key; /* returns the key to be filled in
   the parent */
int last;

• InsertIntoLeaf.c has the following methods

  /* Inserts a key into a leaf node. Returns false
   if the leaf node is full and needs to be
   split. This function only modifies the pageBuf
   if possible else, it returns false. */
int InsertIntoLeaf(pageBuf, attrType, attrLength,
                   value, recId)
char *pageBuf; /* buffer where the leaf page
                   resides */
char attrType;
int attrLength;
char *value; /* attribute value to be inserted*/
int recId; /* recid of the attribute to be
            inserted */

  /* Insert into leaf given the fact that the key
   is old. Used by the InsertIntoLeaf function
   defined above */
InsertToLeafFound(pageBuf, recId, index, header)
char *pageBuf;
int recId;
int index;
AM_LEAFHEADER *header;

  /* Insert to a leaf given that the key is new.
   Used by the insertFunction defined above */
InsertToLeafNotFound(pageBuf, value, recId, index,
                      header)
char *pageBuf;
char *value;
int recId;
int index;
AM_LEAFHEADER *header;
AddToParent.c has the following methods:

```c
/* Adds to the parent attribute value and page number. This method also takes care of the splitting. */
AddtoParent(fileDesc, pageNum, value, attrLength, level)

int fileDesc; /* Necessary to create a new page in case of split */
int pageNum; /* page number to be added to parent */
char *value; /* pointer to attribute value to be added - gives back the attribute value to be added to it's parent */
int level; /* 0 is leaf. Needed to access the rightmost node on the level above */
int attrLength;

/* adds a key to an internal node */
AddtoIntPage(pageBuf, value, pageNum, header, offset)

char *pageBuf;
char *value; /* value to be added to the node */
int pageNum; /* page number of child to be inserted */
int offset; /* place where key is to be inserted */
AM_INTHEADER *header;

/* Split an internal node - populate the new internal node in pbuf1 char* array, and change the header of the internal node that is split appropriately. */
SplitIntNode(pageBuf, pbuf1, header, value, pageNum)

char *pageBuf; /* internal node to be split */
char *pbuf1; /* new internal node's buffer */
char *value; /* pointer to key to be added and to be returned to parent */
AM_INTHEADER *header;
int pageNum;
```
7.1 Summary of code Structure

- Main function insert records into B+ tree using the function insertEntry present in the file InsertEntry.c
- This function creates a new node if the B+ tree is initially empty. Else it calls the insertIntoLeaf function with the following signature:

\[
\text{InsertIntoLeaf}(\text{rightPageBuffers}[0], \text{attrType}, \text{attrLength}, \text{value}, \text{recId});
\]

That is we try to insert in the rightmost node at the leaf level. If this function returns false, we have to split the leaf node.

\[
\text{SplitLeaf}(\text{fileDesc}, \text{pageNum}, \text{attrType}, \text{attrLength}, \text{recId}, \text{value}, \text{last});
\]

- InsertIntoLeaf method checks if there is any space remaining for a new node. If yes it searches for the given key in the leaf to see if it’s already inserted. Based on this it makes either one of the below two calls

\[
\text{InsertToLeafFound}(\text{pageBuf}, \text{recId}, \text{index}, \text{header}) ;
\]

\[
\text{InsertToLeafNotFound}(\text{pageBuf}, \text{value}, \text{recId}, \text{index}, \text{header});
\]

- The above two functions insert into leaf and also modify the header.

- Split leaf function has to handle two cases, one is when there is only one leaf node, and the other is the general case.

- In the first case it creates a new leaf node, and also a root node while in the second case it just creates a new leaf node, and makes a call to the AddtoParent as follows.

\[
\text{AddtoParent}(\text{fileDesc}, \text{rightPageNumbers}[0], \text{value}, \text{attrLength}, 1)
\]

- Since addToParent has to handle the cases of split in parent recursively, it uses the services of two helper functions,
/* Simply add the appropriate fields to the pageBuf */
AddtoIntPage(pageBuf, value, pageNum, header, offset)

/* Split an internal node */
SplitIntNode(pageBuf, pbuf1, header, value, pageNum)

- Whenever the method SplitIntNode is called, we make a recursive call to AddtoParent with the argument level = level + 1

8 Observations
8.1 Statistics Collected

- Number of nodes in the B+ tree
  - We measure this in the pf-layer. In the following function:

```c
PF_AllocPage(fd, pagenum, pagebuf, stats)
int fd; /* file descriptor */
int *pagenum; /* page number */
char **pagebuf; /* pointer to pointer to page buffer*/
int *stats;
/*
   ***********************************************
   SPECIFICATIONS:
   Allocate a new, empty page for file "fd".
   Set *pagenum to the new page number.
   Set *pagebuf to point to the buffer for that page.
   The page allocated is fixed in the buffer.
   ***********************************************
*/
```
Whenever this function is called, a new node is created.

To measure the number of diskReads, diskWrites performed by our program, we had to study and understand the code in pflayer. We observed that the disk is accessed only using two internal helper functions PBufGet, PWritefcn. So, we can keep a counter and measure the number of calls to these functions to measure the above statistics.

- **Number of diskReads**

  ```c
  PFbufGet(fd, pagenum, fpage, readfcn, writefcn, stats)
  int fd; /* file descriptor */
  int pagenum; /* page number */
  PFfpage **fpage; /* pointer to pointer to file page */
  int (*readfcn)(); /* function to read a page */
  int (*writefcn)(); /* function to write a page */
  int* stats;
  /***************************************************/
  SPECIFICATIONS:
  Get a page whose number is "pagenum" from the file pointed by "fd". Set *fpage to point to the data for that page.
  ***************************************************/
  ```

- Whenever this function is called, a disk read operation occurs as the specification suggests. So, we increment diskreads global variable on entering this function.

- **Number of diskWrites**

  ```c
  PWritefcn(fd, pagenum, buf)
  int fd; /* file descriptor */
  int pagenum; /* page to read */
  PFfpage *buf; /* buffer where to read the page */
  /*********************************************/
  ```
SPECIFICATIONS:
Write the page numbered "pagenum"
from the buffer indexed
by "buf" into the file indexed by "fd".
**************************************************************************

– Whenever the above function is called a disk write operation occurs. We write to disk only using this function.

• Number of buffer hits
– We re-access an already written to page using the pflayer method as given below:

PBufGet(fd, pagenum, fpage, readfcn, writefcn, stats)
int fd; /* file descriptor */
int pagenum; /* page number */
PFfpage **fpage; /* pointer to pointer
to file page */
int (*readfcn)(); /* function to read a page */
int (*writefcn)(); /* function to write a page */
int* stats;
**************************************************************************

SPECIFICATIONS:
Get a page whose number is "pagenum"
from the file pointed by "fd". Set *fpage to point to the data for that page.
This function requires two functions:
readfcn(fd, pagenum, fpage)
which will read one page whose number is "pagenum" from the file "fd"
into the buffer area pointed by "fpage".
writefcn(fd, pagenum, fpage)
which will write one page into the file.
**************************************************************************
In this function we have a check to see if the page already exists in our cache. This check is performed as follows:

```c
if ((bpage = PFhashFind(fd, pagenum)) == NULL){
    // cache-miss
} else {
    // cache-hit! yay!
}
```

We maintain a counter to find the number of cache hits in this function. Since we don’t access an already written page using any other function, this successfully measures the number of buffer hits.

- Time.
- kranthi is a good boy.
- We also measure the time using C library function.

```c
gettimeofday(&t0, 0);
```

- We measure time taken to construct the index table.

### 8.2 Control parameters for the Statistics

- We use the following control parameters and measure the statistics by varying them
  - Number of records
  - Cache size (measured in pages)

- We first fix the cache size at a reasonably high value (20 pages). We vary the number of index entries inserted and measure the above defined statistics.

- Next we try out the above method with a low cache size (7 pages).

### 9 Results

- Number of nodes in the case of normal insertion is less as compared to the bulk loading because, in the case of bulk loading each node is tightly packed. Almost all of the nodes (ALL except the rightmost node at every level) are completely full. So, this leads to lower number of nodes. However this is not the case when we insert it normally, since we redistribute whenever a node splits.
- Disk Reads for bulk loading is constant at 1. This disk read corresponds to getting the first page of the index from disk. We needn’t do any disk read after this for bulk loading because, whenever we unfix a page, we don’t have to read that page again. However for normal insertion, we might have to load a page from the disk again even if it is unfixed and evicted from the cache. However, when the buffer/cache size is 20 pages, we always have a cache hit even for normal insertion. This doesn’t hold true when we reduce the cache size as described later in this report.
- No, of disk writes in the case of bulk loading is pretty low compared to the normal loading. In fact bulk loading does the minimum possible number of disk writes. So the difference in the disk writes of bulk loading and normal insertion gives the additional unnecessary disk writes done by the inefficient insertion algorithm. This can be as high as 2000 disk writes for 12.8k records.
- Note that for bulk loading buffer hits is always 0 because we don’t re-access a page after unfixing it. However this may not be the case for normal insertion case. In fact we unfix and reload the root node several times. That is why there may be few buffer hits and few buffer misses.

- Clearly the bulk loading takes less time to populate the index as compared to normal insertion. However, it can also be seen that as the number of records increases, the rate of increase of time is more for normal insertion, as compared to bulk loading. This is because as the number of records increases, number of pages in the B+ tree increases, but the cache size remains the same. So, the cache-hit rate decreases, and the cache-effects begin to wear off, thus resulting in more disk accesses and shoot up in time taken to construct the B+ tree.
## Collected Values

### Performance metrics tables

<table>
<thead>
<tr>
<th>No. of Records</th>
<th>No. of nodes</th>
<th>Disk Reads</th>
<th>Disk Writes</th>
<th>Buffer Hits</th>
<th>Time of insertion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Bulk Loading</td>
<td>B+ tree insertion</td>
<td>Bulk Loading</td>
<td>B+ tree insertion</td>
</tr>
<tr>
<td>200</td>
<td>28</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>37</td>
</tr>
<tr>
<td>400</td>
<td>55</td>
<td>1</td>
<td>1</td>
<td>36</td>
<td>93</td>
</tr>
<tr>
<td>800</td>
<td>109</td>
<td>1</td>
<td>1</td>
<td>90</td>
<td>208</td>
</tr>
<tr>
<td>1600</td>
<td>216</td>
<td>1</td>
<td>1</td>
<td>197</td>
<td>436</td>
</tr>
<tr>
<td>3200</td>
<td>432</td>
<td>1</td>
<td>1</td>
<td>413</td>
<td>892</td>
</tr>
<tr>
<td>6400</td>
<td>864</td>
<td>1</td>
<td>1</td>
<td>845</td>
<td>1907</td>
</tr>
<tr>
<td>12800</td>
<td>1725</td>
<td>1</td>
<td>1</td>
<td>1706</td>
<td>3635</td>
</tr>
</tbody>
</table>

Table 1: These are the list of statistics obtained for cache size of 20. The number of records are varied from 200 to 12800 and each metric values are displayed for insertion using bulk loading and not using bulk loading. Time measured is in micro seconds.

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>Disk Reads</th>
<th>Buffer Hits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bulk Loading</td>
<td>B+ tree insertion</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>39</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: This table contains the different metric values as we vary the cache size. The no. of records are kept constant at 1600
10.1 Remarks on Table 1

- The data used in programmatically plotting the graphs is shown in the table-1 above.
- No. of nodes in the B+ tree obtained by using bulk loading are less than number of nodes in the one obtained by using normal insertion. The difference between them increases with the increase in no. of records.
- The disk read metric values are 1 for both the methods because of the reason mentioned above.
- The number of disk writes in case of bulk loading is less as compared to normal insertion and the difference between them increases with the number of records.
- There will be no buffer hits at all in case of bulk loading while in case of normal insertion there will be and the number of buffer hits increases with number of records inserted.
- The time of insertion in case of bulk loading is lesser as compared to normal insertion and the difference between them increases with increase in number of records.

10.2 Remarks on Table2

- As we decrease the cache size the disk reads in case of normal insertion increases.
- This is because, in the case of normal insertion, every time we insert a node, we start from the root and fetch every node in the path from the root to the leaf containing the required index entry.
- This requires fetching at least height of the B+ tree number of nodes(typically 3). When the cache size is 20pages, these nodes are directly fetched from the cache and not from the disk.
- That is why the number of disk reads in the table-1 is always 1. However as we begin to reduce the cache size, lesser and lesser B+ tree nodes have a cache-hit and this results in more disk reads. However bulk loading always requires 1 disk read(first page).
11 Conclusions

- From the above observations, it is very clear that, bulk loading is better than normal insertion in every possible way.

- Hence it is always desirable to bulk load data if it is available in sorted order, or if it can be sorted quickly and in an efficient manner.

12 Future Scope of Work

- Bulk loading can also be implemented using a bottom up fashion from the leaf node to the root accessing only one level at a time.

- One could then compare the statistics and decide which way would be better for bulk loading.

13 References

- Database System Concepts taught in class and text reference textbook by Abraham Silberschatz, Henry F. Korth, and S. Sudarshan

- Bulk loading in https://en.wikipedia.org/wiki/B