CS174B Database Final Project Report

Shaoyi Zhang
Nuan Wen
Zhancheng Qian

Winter 2018
University of California Santa Barbara

Distribution List

Professor

Xifeng Yan

Teaching Assistant

Yi Ding

Dev(s)	Shaoyi, Zhang
	Nuan, Wen
	Zhancheng, Qian
Test	
Last Updated	March 18, 2018
Document Status	Final
Target Release	
Approvers	

Contents

CS174B Database Final Project Report

Shaoyi, Nuan, Zhancheng
University of California, Santa Barbara

March 18, 2018

Abstract

This a documentation for the design & implementation of CS174B Database Final Project. This project aims for extracting DOC ids and generating a word index file. Then bulk load the index file into B+ tree for quick access, insertion & deletion of elements.

1 Introduction

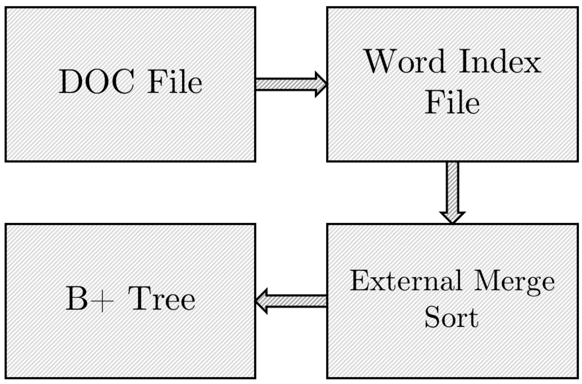


Figure 1: The overall structure and relationship between each component

1.1 Overall Structure

There are multiple components in the workflow. The first step is to read the DOC File and transform into word index file. The second step is to read the word index file and use external merge sort to sort multiple index files and merge them into one. The third step is to read the merged index file and do bulk loading into B+ tree so that we can do access, insertion & deletion.

1.2 Task & Duties

The task here are divided evenly to three members in the team. The first step from DOC File to Word Index is implemented and tested by Nuan Wen. The second step of external merge sort of the multiple index files is implemented and tested by Zhancheng Qian. The third step bulk loading the external merge sort into the B+ tree for easy data access is implemented and tested by Shaoyi Zhang.

In total:

- ☞ Nuan Wen: Implement and test build index
- ☞ Zhancheng Qian: Implement and test External Merge Sort
- ☞ Shaoyi Zhang: Implement and test bulk loading B+ tree

2 First Step - Reading Raw & Generating Index File

The first step: sequentially reading data from the DOC File and generating index file which contains in each row a word and its corresponding doc IDs. And there will also be a file that records the line number of each word in the index file, which can facilitate the bulk loading process.

Index File	Word And Its Doc Ids
Word Line File	Line Number in Index File

2.1 Design of the *Index File Generator*

The *Index File Generator* has function of *buildIndex*

```
void buildIndex(const string& textFile, const string& indexFile, unsigned int ...
    map<string, unordered_set<unsigned int>> index;
        ...
    string line;
    while (getline(infile, line)) {
        istringstream iss(line);
        vector<string> words{istream_iterator<string>{iss}, istream_iterator...
        string doc = words[0];
        unsigned int line_no = stoi(doc.substr(3, doc.length()));
        for (vector<string>::iterator iter = words.begin() + 1; iter != words.end()...
            if (sizeof(iter)>=256) continue;
            index[*iter].insert(line_no);
     ...
```

Listing 1: BuildIndex implementation in a nut shell

We utilized C++ standard library to optimize our performance. Specifically we used map to store `string` and `unordered_set` pairs, so that we can directly output them into the output index file. And we also used `unordered_set` to remove possible duplicate line numbers in the output index file, while maintaining the performance, since it doesn't check order information.

```
while (idPrinted < totalSize) { // checking if the output is greater than pagesize
        if (currSize < pageSize)
                // do things if the output is smaller than pagesize
        else
                // do things if the output is greater than pagesize
```

Listing 2: PageSize checking

3 Second Step - External Merge Sort of Index File

The Second step: Due to the huge size of the index file, we shall use external merge sort to sort and merge the index file. The preprocess starts with slicing the big index file into multiple parts that can fit in the memory buffer and sort it using merge sort. The implementation of external merge sort involves using a `minHeap` and also reading lines from index files and inserting them into the `minHeap`.

3.1 Design of the *External Merge Sort*

The *External Merge Sort* has implementation of a `minHeap`

```
class MinHeap {
        MinHeapNode *harr;   // pointer to array of elements in heap
        int heap_size;   // size of min heap
        int capacity;
public:
        MinHeap(MinHeapNode a[], int size);
        void MinHeapify(int);   // to heapify a subtree
        int parent(int i);   // to get index of parent
        int left(int i);   // to get index of left child
        int right(int i);   // to get index of right child
        MinHeapNode getMin();   // to get min of the minHeap
        void replaceMin(MinHeapNode x);   // to replace the min
        void print();
        void insertKey(MinHeapNode k);
};
```

Listing 3: minHeap implementation of EMS

We utilized the `minHeap` to store some sorted information of index records reading from multiple index files. This is very efficient time-wise. And it can adapt to our design very well. Each time the program gets an index record from the `minHeap`, we output it into the output, and read another index record from the index file to put it into the min position of the `minHeap`, and then minHeapify it.

```
for (i = 0; i < run_size; i++){ // writing lines from index file into multiple files
        if (!getline(in, line)){
                more_input = false;
                break;
        }
        vector<std::string> tokens;
        vector<unsigned long> tmp_vec;
        boost::split(tokens, line, boost::is_any_of("\t"));
        arr[i].word = tokens[0];
        for (int k = 1; k < tokens.size(); k++)
                tmp_vec.push_back(stol(tokens[k]));
                arr[i].vec = tmp_vec;
        }
        mergeSort(arr, 0, i - 1); // sort array using merge sort
```

Listing 4: Spliting index files

We first created multiple smaller index files by just splitting the original big index files, based on the rule of reading as much as the memory buffer size allows. And also do the merge sort on each small index files. This is the preprocess paving way for the actual external merge sort.

```
while (count != i) {
        MinHeapNode root = hp.getMin();
        string s = root.element.word + "\t";
        std::vector<unsigned long>::iterator it;
        for (it = root.element.vec.begin(); it != root.element.vec.end(); ++it) {
                ...
        string line;
        if (!getline(in[root.i], line)) {
                vector<unsigned long> vec; vec.push_back(0);
                struct IndexRecord ir = {"", vec};
                root.element = ir;
                count++;
        }
        hp.replaceMin(root);
```

Listing 5: Getting and replacing min element

This part is about getting the min from `minHeap`, and place the next record into the `minHeap`. This loops until all smaller index files run out of index records.

4 Third Step - Constructing B+ Tree

The Third step: Using Bulk-loading to construct the B+ Tree in order to achieve quick access, insertion and deletion of records. We implemented *load*, *insert*, *delete* of record into the B+ Tree. The overall structure of the B+ Tree looks like:

```
class BPlusTree{
private:
        Node * root;
        int count;
        long maxPage;
public:
        BPlusTree();
        BPlusTree( long maxPage );
        BPlusTree( string word, FilePointer record, long maxPage );
        ~BPlusTree();
        int getCount(){ return count; };
        bool isRoot( Node * cur ) const { return cur == root; }
        void insert( string word, FilePointer record );
        void insert( Node * parent, Node * child );
        bool remove( string word );
        Node * insertHelper( string word ); // find internal node candidate
        void splitNoneLeaf( Node * cur );
        void splitLeaf( Node * cur );
        void splitRoot( Node * cur );
```

Listing 6: Overall structure of B+ Tree Class

4.1 Implementing the *Builk Load*

The *Builk Load* loads entries into the B+ Tree

```
void BPlusTree::bulkLoad() {
        Node * right = root;//root->getChildAt(1);
        for ( auto it = dirPage.begin(); it != dirPage.end(); ++it ) {
        right = this->insert(it->first,
                FilePointer( it->first, it->second, it->first, 0 ),
                right)->getParent();
     ...
```

Listing 7: BuilkLoad implementation

The *Builk Load* function is implemented as sequentially loads parts of data from the index file, and put each entry at the rightmost position of the B+ Tree. Repeat this process until we get the entire index file into the B+ Tree.

4.2 Implementing the *Insert Function*

The *Insert Function* inserts entries into the B+ Tree

```
void BPlusTree::insert( Node * parent, Node * child ) {
        this->levelOrder( child );
        this->levelOrder( parent->getParent() );
        this->levelOrder( root );
        int index = parent->indexOfChild( child->getKeyAt(0) );
        if ( parent->size() == parent->childSize() ) {
                parent->insertChild( child );
        }
        else {
        index++;
        for (int i = parent->size(); i >= index;i--){
                parent->setKeyAt( i, parent->getKeyAt( i - 1 ) );
                parent->setChildAt( i + 1, parent->getChildAt( i ) );
        }
        parent->setChildAt( index, child );
        ...
```

Listing 8: Insert Function Implementation

The *Insert Function* is implemented taking one given record and try to insert into the B+ Tree. Depending on different cases, this might involve split of some nodes in the B+ Tree. Thus the implementation of this *Insert Function* also requires the additional implementation of other functions such as the *splitRoot Function* and the *splitNoneLeaf Function*.

4.3 Implementing the *Remove Function*

The *Remove Function* deletes entries from the B+ Tree

```
bool BPlusTree::remove( string word ) {
        Node * parent = insertHelper( word );
        int index = parent->indexOfChild( word );
        Node * leaf = parent->getChildAt( index );
        if ( leaf == nullptr )
                return false;
        leaf->removeKeyValuePairAt( indexOfKey( word ) );
        Node * left = leaf->getPrev();
        Node * right = leaf->getNext();
        if ( leaf->size() > L ) {
                parent->setKeyAt( parent->indexOfKey( leaf->getKeyAt(0) ) );
        } else if ( left != nullptr ) {
                if ( left->size() > L ) {
                ...
```

Listing 9: Remove Function Implementation

The *Remove Function* is implemented using lazy deletion. Marking the record as deleted and skip when traversing next time. The design choice of this is that there will not be much deletion. And this is also because the deletion in B+ Tree cost large amount of running time, and it is not efficient in large scale database.

5 Page Layout & Buffer Management

5.1 Page Layout

The Page Layout of our design is in two format.

First is that in the index file, we treat one page as the following:

```
Word, DOC_ID1, DOC_ID2, DOC_ID3, DOC_ID4 ...
```

The DOC_IDs points to each DOC that contains the word.

Second is that in the word line file, we treat one page as the following:

```
Word, LINE_NUM ...
```

The LINE_NUM here points to which row in the index file that contains word.

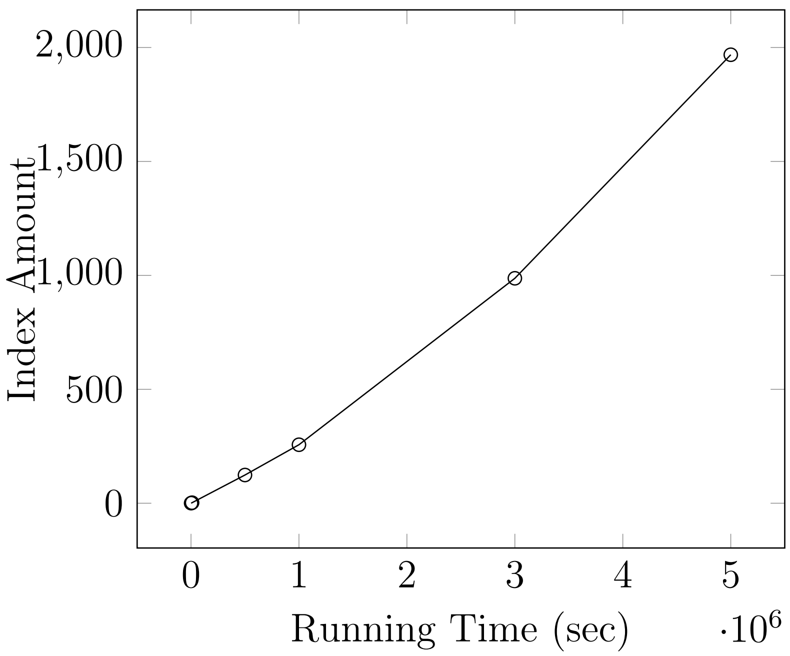
5.2 Buffer Management

We implemented the Buffer Management in two places, without having one single Buffer Manager, we put the feature of Buffer Management into the External Merge Sort and also the Bulk Loading. In the External Merge Sort, we implemented it as there is a buffer size such that we can read in lines of index file until the buffer is full. Then we process the data and read in the next batch amount of data. In the Bulk Loading of B+ Tree, we have similar measures which read in data until the buffer is filled, then we write that data into the B+ Tree, then we continue reading more data into the buffer, and repeat this process.

6 Experiment on the Performance

6.1 Build Index Performance

Build Index Performance	
Index Amount	Time (sec)
1000	0.29034
10000	2.29788
500000	123.802
1000000	256.682
3000000	987.093
5000000	1968.15



6.2 B+ Tree Performance

B+ Tree Performance	
Index Amount	Time (sec)
1000	15.3029
2000	35.4120
4000	85.1245
8000	203.2913
10000	280.9102