## Description

The program take an input of TREC format document and build an inverted index table, along with a lexicon file and doc table in a I/O and memory efficient manner.

After a successful build, you can query terms(**alpha or number considered**) in the established index database.

Example TREC file: https://github.com/microsoft/TREC-2019-Deep-Learning.

## Usage

### Make

```
$ make # use this to compile the project
```

The executable will be created with the name 'main' under root directory. Run by ./main [options]

```
$ make clean # You can use this to clear all the output artifacts if necessary.
```

### Usage

```
-b <TREC file>  Build inverted index table, lexicon and doc table from input
file(without <>). Output to output/... \n
[-p <1~10>] Load from 10 to 100 percent of data, mapping to 1 to 10 to 10% to
100%. Default to 10(100% data)
[-m <400~8000>] Memory Limit 400MB to 8GB, mapping to 400 to 8000\n . Default
to 1000(1G)
-q [TREC file] Query mode. TREC file is optional if you want to show snippet.

[] means optional.

$ ./main -b msmarco-docs.trec -m 1000 # build and set memory limit to 1G (not
accurate)

$ ./main -b msmarco-docs.trec -p 1 # build upon only 10% of data

$ ./main -q msmarco-docs.trec # enter query mode, show snippet (after you
successfully built)
```

**Output**:

During building process, some temporary files will be created in ./tmp/ , but they will be **deleted automatically** after successful build.

After being successful built, final artificats(an inverted table, a lexicon and a doc table) will be stored in output/ . Deleting any file under this directory will require new build.

# How it works

Building process is broken down to four phases: parse (main.c)--> sorting (sort_phase.c)--> merging (merge_phase.c)--> final building(final_build.c). Every phase have their intermediate input and output files so they are loosely coupled.

## Parse phase (main.c)

**Input:** Trec file

**Output:** N intermediate temporary file(./tmp/intermediate-x)  and a term id <--> term map file (**./tmp/tmp_term**)

```
// main.c

/*  Hash Map  */
/* store mapping <term_id, term> during parsing, to ensure same string map to
the same id */
struct hashmap *term_map;

typedef struct
{
   char key_string[KEY_MAX_LENGTH];
   int number; // term id
   term_entry* te;
} map_entry;

uint64_t user_hash(const void *item, uint64_t seed0, uint64_t seed1);
int user_compare(const void *a, const void *b, void *udata);

/* map-iterator callbacks */
bool handle_map_entry(const void * value, void * fb);
bool _free(const void * value, void * x);

/* End of hash map */

/*  Document Parser  */
void parse(FILE* f_in, int percentage);
int process_document(int len, char* buf, long doc_offset);
void process_term(int len, char* term, int doc_id, int pos);
/*  End of Document Parser  */

//termid.h
#ifndef TERM_ID_H
#define TERM_ID_H
```

```
#include <stdio.h>
#include "model.h"
#include "hashmap.h"

/*  global term<-->term_id map(./tmp/tmp_term) Generated in parse phase, used
in sort, merge phases for intermediate process.*/
extern char** term_id_map;

void init_term_id_map();
void flush_term_to_file(bool end, int total);
void write_term_id_map(int term_id, char* term, int len, int total);
char** read_term_id_map(int* retlen);
void free_term_id_map(char** tm, int len);

#endif
```

## Step:

1. Read in document with a buffer and parse like stream.
2. Parse the document between <TEXT> tags. First line is URL. Parse the rest to extract out words. Use a hashmap(term string as key) and a counter to assign each distinct word(uncapitalized during parsing) a term id, and store its doc-ids and frees in the hashmap in the form of linked-list node structure as value:

```
/*  model.h */
/*  posting model  */
//  TE(term_id) -> TC(doc_id)

// Term Entry Chunk, by doc_id
typedef struct TC{
    int doc_id; // doc id
    int freq;   // frequency of words in this doc
    struct TC* next; // next TC Node
}term_chunk;

// Term Entry
typedef struct TE{
    int total_size;
  // total size of this TE node = sizeof(term_id) + sizeof(total_sizes) +
sizeof(tc_length) + sum(full size of each TC node)
    int term_id;   // term id
    int tc_length; // TC list length
    struct TC* chunk_list_head; // TC list head
    struct TC* chunk_list_tail; // TC list tail
}term_entry
```

Whenever encountered a word,  query if is already in the hashmap.  If not, create a TE node and its child TC and child PN for it and put the TE node in the map.  If yes, retrieve the existing TE node and check the doc-id of its TC list tail(chunk_list_tail), because doc-id is sequentially assigned, so if the tail TC has same doc-id as now --> create a PN node and append to the TC, if the tail TC has a different(must be smaller) doc-id --> create a new TC node and child PN,  append it to the TE.

You can see the above structure does not contain the term itself.  The actual entry node stored in hashmap is as follows:

```
/*  main.c  */
/*  Hash Map  */
/* store mapping between term_id and term during parsing, to ensure same string
map to the same id */
struct hashmap *term_map;

typedef struct
{
   char key_string[KEY_MAX_LENGTH];
   int number; // term id
   term_entry* te;
} map_entry;
```

3. The hashmap will grow very large over time. So set a limit for the memory of hashmap. If exceeded, writing the TE node information of every node in hashmap to a **separate intermediate file** while keeping map_entry node because we still need it for distinct-word purpose, yet we can free the TE node after writing out.

   **Intermediate posting file format:**

   (Length{4 bytes}, Term ID{4}, Doc List Length{4}, [ (Doc ID{4}, Freq{4}) , (Doc ID{4}, Freq{4}) ,...] ) ( ... ) ...

   - **Length:** the total length of this unit.
   - **Doc List Length:** number of doc IDs in its TC list
   - **Freq:** frequencies of the word in this document

   So it is just a little-compressed format of TE node structure. The space efficiency is not good because every word will at least take 4 bytes in this format so in general the result inverted table will be as large as original file. It is something I need to compress in the next assignment.

So in the end, there will be many intermediate files(0 - n). If i > j,  doc-ids in inter-i are larger than doc-ids in inter-j.

4. Term <--> Term ID map (term_id.c):

   This mapping can be output(to a buffer, then flush to a file if full) every time encounter a new word.

   **Term id map file format:**

Total Terms{4},  (Term ID{4}, Term Length{4}, Term{determined by Term Length})  ( ... ) ...

- **Total**: total number of terms. This is for convenience of reading because the reader end can know the length of array it needed to store all the records.
- **Term Length:** Length of the term string, which determines the length of the Term field.

5. Doc table

   Doc table is output during document parsing with a standalone output buffer.

   **Doc table format**

   Total Size{4}, (Unit Length{4}, Doc_ID{4}, Doc_Size{4}, Offset{8} , URL{determined by Unit Length}) (...) ...

   **Doc_Size:** length of original document

   **Offset**: offset in original TREC file, used to generate snippet

## Sort phase (sort_phase.c)

**Input:** N intermediate files, tmp_term file(containing the map between term id and term)

**Output:** N sorted intermediate files

Read in each intermediate file and transform the binary into struct model described earlier for convenience of sorting. Read in term id map into an array instead of a hashmap, because only  id --> term is needed during sorting. Also in this step I assume the memory can fit in one intermediate file.

## Merge phase (merge_phase.c)

**Input:** N sorted files, term id mapping already readed during sort phase

**Output:** 1 merged file containing sorted and no duplicate term-id record

Use merge sorted lists algorithm w/ heap, but improves the I/O efficiency by assigning a buffer for each input files and final output.

Also, in this step we can easily merge every two record with the same term id which were dispersed across different intermediate files in the first phase because of the sorted nature of each input file and the usage of heap. So aftter this step, we have 1 merged  and sorted file with no duplicate term-id record.

## Build phase (final_build.c)

**Input:** single merged intermediate file, term id mapping file

**Output:** 1 inverted index file, 1 lexicon , 1 doc table(already created in phase 1 )

## Compression(var_bytes.c)

The final inverted index is compressed by **simple var bytes compression** and the document id also use **difference compression**.

```c
// var_bytes.c
#include <stdlib.h>
#include <stdio.h>

unsigned char* vb_encode(int n, int* L ){
    int m = n;
    int d = 0;
    if( m == 0 )
        d = 1;
    while(m != 0){ m/=128; d++;}
    unsigned char* bytes = (unsigned char*)malloc(d);
    for(int i = 0; i < d; i++, n /= 128)
        bytes[i] = n % 128;

    bytes[d-1] += 128;
    *L = d;
    return bytes;
}

int vb_decode_stream(FILE* f){
    int r = 0,n = 0, k = 1;
    unsigned char c = 0;
    do{
        fread(&c, 1, 1, f);
        n = c % 128;
        r = n*k + r;
        k = k*128;
    }while(c < 128 && !feof(f));
    return r;
}
```

## Format

### Notation

Since the final file format become more complicated,  I define a format notation:

```
>> {top level structure S}
  - substructure _S1 of S
    : description
    >> {structure of substructure1}
      - substructure __S1 of _S1
      ...
  - substructure _S2 of S
    ...
```

## Inverted index file format

Blockwise structure. And all data are var-byte compressed(var). The final index file is only 1.72 GB.

```
>> {term1_inverted_list, term2_inverted_list ... }
  - term_inverted_list
    : Inverted list for term T, its start offset is mapped in lexicon file.
    >> {metadata, last_docid_list, block_size_list, block1, block2 ...}
        - metadata
            : Metadata of each inverted list of term T
            >> {total_docs(var), last_docid_list_size(var),
block_size_list_size(var) }
              - total_docs: Number of documents containing this term
              - last_docid_list_size: Total bytes of the last docid list.
Because every element in this list                  is also var-byte
compressed, so we need a total length in meta.
              - block_size_list_size: same reason as above.
        - last_docid_list:
            : Last docid of each block.
            >> {last_docid_of_block1(var), last_docid_of_block2(var), ...}
        - block_size_list
            : Block size (in bytes) of each block. Because every docid or freq
is also var-byte compressed.
            >> {block_size_of_block1(var), block_size_of_block2(var), ...}
        - block
            : Why there isn't a doc block size in meta to separate doc and
freq? Because we know last doc.
            >> {doc1(var),doc2(var), ... doc128, freq1(var),freq2(var), ...
freq128}
```

## Lexicon file format

Lexicon file and doc table can also be easily var-byte compressed similarly as index file. But considering they are not very large, I choose to skip compression until I have time.

```
>> {term1, term2, ... }
  - term
    >> {Term Length(4), Term(length is determined by Term Length), Offset(8)}
      - offset: The offset in the inverted list of this term.
```

## Doc table format

```
>> {metadata, doc1, doc2, ...}
  - metadata
    >> {Total Doc(4)}
      - Total Doc: the total number of documents
  - doc
    >> {Unit Length(4), Doc_ID(4), Doc_Size(4), Offset(8), URL(determined by
Unit Length)}
      - offset: offset in original TREC file, used to generate snippet
```

## Implementation

```
// model.h , implemented in model_supportted.c

/*  doc output  */
bool write_doc_table(file_buffer* fb, doc_entry* de, bool flush);
bool next_doc(file_buffer* fb,  doc_entry* de);
void write_doc_table_end(file_buffer* fb, int total);

/*  inverted list op  */
// blockwise structure, document id is diff-compressed
#define BLOCK_SIZE 128
long write_to_final_inverted_list(file_buffer* fb, term_entry*te, bool flush);
term_chunk* read_block_to_cache(IV* lp);

// lexicon.h, implemented in lexicon.c

#ifndef LEXICON_H
#define LEXICON_H

#include <stdio.h>
#include "hashmap.h"

// word length limit
#define WORD_LENGTH_MIN 1
#define WORD_LENGTH_MAX 15

/* Lexicon model */
typedef struct{
    int term_length;
    char* term;
    long offset; //offset in inverted list
}lexicon;

void write_lexicon_file(FILE* f,lexicon* lex);
bool read_lexicon_file();
void close_lexicon_file();

#endif
```

# Query(query.c + lexicon.c)

## Preparing

1. Read in the whole lexicon file to a hashmap.
2. Read in the doc table to an array structure.

```
/*  query.h  */
#ifndef QUERY_H
#define QUERY_H
#include "lexicon.h"

typedef struct
{
   char key_string[WORD_LENGTH_MAX+1];
   long offset;
} lexicon_el;

#define CONJUNCTIVE_MODE 0
#define DISJUNCTIVE_MODE 1

bool init_query_database()
bool query(char** s, int N, int limit, int MODE, char* doc_path);
void close_query_database();

#endif
```

```
// query.c

struct hashmap * lexicon_map;
doc_entry** doc_table; // doc_entry array

bool init_query_database(){
    printf("Preparing...Do not enter anything\n");
    lexicon_map = hashmap_new(sizeof(lexicon_el), 0, 0, 0,
                                l_hash, l_compare, NULL);

    // read lexicon file to lexicon_map, read doc_table file to doc_table
array
    if(read_lexicon_file(lexicon_map) == false || read_doc_file() == false)
        return false;
    printf("Done.\n");
    return true;
}
```

## Query Engine

### DATT Interface

```
// inverted_list.h

#ifndef IV_H
#define IV_H

#include "model.h"

int nextGEQ(IV* lp,int k);
int getFreq(IV* lp);
IV* openList(int offset, int maxdocID);
void closeList(IV* iv);
void free_cache(IV* iv);

#endif

// model.h
typedef struct iv{
    FILE* f;
    int length;
    long block_ldoc_table_offset;
    long block_size_table_offset;
    long block_offset;
    term_chunk* block_cache; // block is cached
    term_chunk* block_cache_ptr;
    int block_curr_ldoc;
    int block_curr_sdoc;
    int block_curr_cnt;
    int curr_doc;
    int curr_freq;
    int maxdocID;
}IV;
```

## Query(query.c)

### Pseudo Code

```
LP = []
rank_heap = []

# open lists
for every term in terms:
  LP[i] = openList(term)

# sort to start from shortest list
sort(LP, lambda lp: lp->doc_length)

# two modes
switch MODE:
  CONJUNCTIVE: conjunctive_query(LP, &rank_heap)
  DISJUNCTIVE: disjunctive_query(LP, &rank_heap)

# results
TOP_K = print(rank_heap)
```

## Conjunctive Query

Use DATT to try skip all the lists to the same doc id.

### Pseudo Code

```
rank_heap
did = 0
d = 0
while did < maxdoc:
  did = LP[0].nextGEQ(0)
  if did >= maxdoc:
    break
  i = 1
  while i < cnt and (d = LP[i].nextGEQ(did)) == did: # skip
    i++
  if d > did: # fail AND, advance with max
    did = d
  else:        # success AND, calculate BM25 and add to rank heap
    bm25 = 0
    for i in (0...len(LP))
      # BM25 calculation: all parameter are known
      bm25 += calBM25( fdt = LP[i].getFreq(), ft = LP[i].doc_nums, d =
getDocSize(did))
    if rank_heap.size < K:
      rank_heap.add({bm25, did})
    else:
      if bm25 > rank_heap.top().bm25:
        rank_heap.pop()
        rank_heap.add({bm25, did})
    did++
```

## Disjunctive Query:

Use two min heaps. One heap with bm25 as key, doc_id as value for Top-K.  Another heap with doc_id as key, lp as value to repeatedly extract the smallest doc id of all term lists.

### Pseudo Code

```
LP = [...]
LP_heap
rank_heap
for every lp in LP: # initialize LP_heap
  doc = lp.nextGEQ(0)
  LP_heap.add({doc, lp})

while LP_heap:
  top_doc = LP_heap.top().doc
  bm25 = 0
  while LP_heap.top().doc == top_doc: # pop out all the same smallest doc id
    pop = LP_heap.pop()
    bm25 += calBM25(pop)              # calculate BM25
    nextdoc = pop.lp.nextGEQ(top_doc) # get next doc and push back to the heap
    if nextdoc < maxDoc:
      LP_heap.add({nextdoc, pop.lp})

  # It is obvious that this smallest doc will never be accessed again, so feel
safe to add to heap
  if rank_heap.size < K:
    rank_heap.add({bm25, top_doc})
  else:
    if bm25 > rank_heap.top().bm25:
      rank_heap.pop()
      rank_heap.add({bm25, top_doc})
```

## Snippet Generation

Simple method. Use a fixed size sliding window, the score is based on term match count.

## Buffer

Implemented a structured dynamic buffer because there are many cases that need buffer. We can bind a file pointer, buffer array together to ease the management.

```
/*  model.h */
/*  Dynamic Buffer  */
typedef struct{
    FILE* f;
    char* buffer;
    int size;
    int curr;
    int max;
}file_buffer;

/*  Operate the term model by dynamic buffer */
file_buffer* init_dynamic_buffer(int size);
void free_file_buffer(file_buffer* fb);
bool next_record_from_file_buffer(file_buffer* fb, term_entry* te);
bool write_record(file_buffer* fb, term_entry* te, bool flush);
void flush_buffer_to_file(file_buffer* fb);
```

## Example

```
$ ./main -b ~/GoogleDataSet/msmarco-docs.trec
|       Progress        |       Identified Terms        |
|       0.43    %       |       369658                  |
|       0.87    %       |       611716                  |
|       1.30    %       |       825976                  |
|       1.74    %       |       1011280                 |
|       2.17    %       |       1190474                 |
|       2.61    %       |       1359676                 |
|       3.04    %       |       1517451                 |
|       3.48    %       |       1684005                 |
|       3.91    %       |       1827872                 |
|       4.34    %       |       1972042                 |
Successfully created tmp/intermediate-0...
|       4.78    %       |       2122592                 |
|       5.21    %       |       2268322                 |
...
|       94.70   %       |       18990120                        |
Successfully created tmp/intermediate-20...
|       95.13   %       |       19049480                        |
|       95.57   %       |       19112203                        |
|       96.00    %       |       19173850                                 |
|       96.44    %       |       19234465                                 |
|       96.87    %       |       19298141                                 |
|       97.31    %       |       19361124                                 |
|       97.74    %       |       19423488                                 |
|       98.18    %       |       19486716                                 |
|       98.61    %       |       19550311                                 |
|       99.04    %       |       19608970                                 |
|       99.48    %       |       19682976                                 |
Successfully created tmp/intermediate-21...
|       99.91    %       |       19745774                                 |
|       100.00   %       |       19757161                                 |
Successfully created tmp/intermediate-22...
Total terms: 19757161
Total documents: 3213609
Preparing for sort...
Sorting...
Merging sorted files...
Successfully merged to file tmp/merged-0
Building final files...
All success. You can use -q to query words.
```

```
$ ./main -q ~/GoogleDataSet/msmarco-docs.trec
Preparing...Do not enter anything
Done.

Enter result limit:
10
Choose mode: [0] Conjunctive mode. [1] Disjunctive mode.
0

Enter your query:
hello world

 ----------------------------------------
|        Showing Top 10 Result           |
 ----------------------------------------

[1]     http://www.scriptol.com/programming/hello-world.php
BM25:   19.963130
<Snippet>
xt/css" ?> <window
xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul"
align="center" pack="center" flex="1"> <description>hello, world!</description>
</window>another smaller collection of hello world programs.
but this is a joke at gnu.
sieve of eratosthenes in any programming language.
```

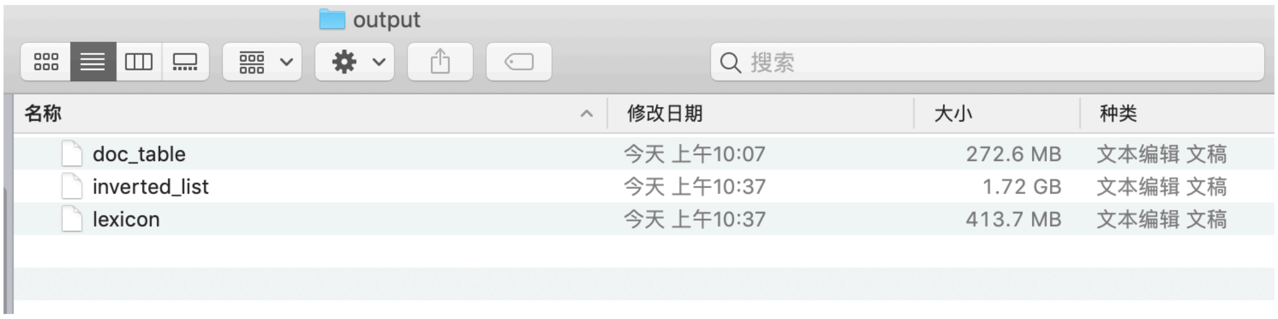
```
fibonacci numbers in any language.
submit more listings in comments.
created in 2006.
last modified april 10, 2013 this page is free to print and copy for any usage.
don't use it on  another website (dup
</Snippet>

[2]     https://en.wikibooks.org/wiki/Computer_Programming/Hello_world
BM25:   19.822599
<Snippet>
000010000000000100000100000
00010000010000100000000100000000000100000000000100000100000
00010000010000100001000100000000000100000000000100000100000
00111001110011111110011111111000111111111000001111000000
00000000000000000000000000000000000000000000000000000000  enddraw endsm  [
edit]the oo language looks like c.use system.
windows.
forms;class hello
world extends system.
windows.
forms {protected:
string hw;construct hello
world  () {this.
hw = 'hello, world!
';}public void function show  () {message
box.
show
</Snippet>

...

[10]    https://msdn.microsoft.com/en-us/library/ms235631.aspx
BM25:   19.209816
<Snippet>
; // convert a wide character system::
string to // a wide character cstring
w string.
cstring
w cstringw (wch); cstringw += " (cstring
w)"; wcout << (lpctstr)cstringw << endl; // convert a wide character system::
string to // a wide character basic_string.
wstring basicstring (wch); basicstring += _t (" (basic_string)"); wcout <<
basicstring << endl; delete orig; }output
hello, world!
(system::
string) hello, world!
(char *) hello, world!
(wchar_t *) hello, world!
(_bstr_t) hello, world!
```

```
(ccom
bstr) hello
</Snippet>
```



| 名称 | 修改日期 | 大小 | 种类 |
|---|---|---|---|
| doc_table | 今天 上午10:07 | 272.6 MB | 文本编辑 文稿 |
| inverted_list | 今天 上午10:37 | 1.72 GB | 文本编辑 文稿 |
| lexicon | 今天 上午10:37 | 413.7 MB | 文本编辑 文稿 |

## Result

On input of 22GB TREC file

**Option**:

```
-m 1000 : approxi~ limit memory to 1GB ~ 1.5GB
```

**Word length limit** setting in lexicon.h :

```
#define WORD_LENGTH_MIN 1
#define WORD_LENGTH_MAX 15
```

**CPU:** 2.7 GHz Intel Core i7

**Parse phase speed:** ~40 minutes

**Total terms**:  19757161

**Total documents**: 3213609

**Sort phase :**  7 minutes

**Merge phase :** 4 minutes

**Build phase:**  8 minutes

**Inverted Index File Size:** 1.72 GB

**Lexicon File Size:**  413.78 MB

**Doc Table Size:** 272.6 MB

**Query Preparing :** 12 seconds

**Query Speed**: Nearly instant ( < 100ms) for conjunctive or disjunctive mode , even with > 100 words

## Memory Check

```
==54418== LEAK SUMMARY:
==54418==    definitely lost: 0 bytes in 0 blocks
==54418==    indirectly lost: 0 bytes in 0 blocks
==54418==      possibly lost: 8,840 bytes in 7 blocks
==54418==    still reachable: 14,880 bytes in 162 blocks
==54418==         suppressed: 0 bytes in 0 blocks
```

## Limitation

In blockwise structure of inverted list, those very short list with << 128 doc ids can be further optimized.

Doc table and lexicon can also be var-byte compressed.