Report for Bitmap Indexing Merge Using 10Mb-Memory and RLE Compression

Tianlin Yang 40010303 Peiqi Wang 40050030 Zelan Xiao 40059394

April 14, 2020

Abstract

This report investigates bitmap indexing technique for tuples data duplicates elimination and sorting merge. First, for a given T1.txt and T2.txt merge task, our implemented code has been generated using eclipse. The test of our code shows the same result with the original Lab1 result. Second, we improved some parts of our project. The compression has been implemented via EWAH and RoaringBitmap but not suitable for ID bitmap sorting so finally RLE is be used. And speed-up encode-decode feature for less disk I/O has been established. After that, the problem we found in the bitmap indexing is when 10mb memory limitation, it can't avoid split processing, therefore we researched more materials that are related to the speed up, and we dug out one of the reliable solutions is using bitset in Java. Finally, we compared the result of these things.

1 Introduction

In this report we mainly consists of two parts: Bitmap indexing merge implementation, space and time tests.

Currently, bitmap indexes have been considered to work well for low-cardinality columns, which have a modest number of distinct values, either absolutely, or relative to the number of records that contain the data. Bitmap indexes have a significant space and performance advantage over other structures for query of such data. Their drawback is they are less efficient than the traditional B-tree indexes for columns whose data is frequently updated: consequently, they are more often employed in read-only systems that are specialized for fast query - e.g., data warehouses, and generally unsuitable for online transaction processing applications. In our project, we are focus on the ID bitmap domain. It can provide the duplicate locations by generate bitset in Java, which not increase too much cost for both memory and disk I/O. There are many methods already be used for bitmap compression. JavaEWAH[2] is one of the high speed methods, it shows much stronger performance than others. Researchers did some experiments to test the bitmap[3] robustness, and from all of the papers we could find an similar result: when the bitset approach is applicable, it can be orders of magnitude faster than other possible implementation of a set (e.g., as a hash set) while using several times less memory.

The purpose of an index is to provide pointers to the rows in a table that contain a given key value. This is achieved by storing a list of rowIDs for each key corresponding to the rows with that key value. Each bit in the bitmap corresponds to a possible rowID, and if the bit is set, it means that the row with the corresponding rowID contains the key value. A mapping function converts the bit position to an actual rowID. For instance,Figure.1is an example of bitmap indexing.

2 Implementation

In this section we implemented multiple algorithms and which be used for test speed and disk I/O on different inputs and use running time and cost score respectively to evaluate the performance later.

Name	Gender	City
Justin	M	Vancouver
Jane	F	New York
Gigi	F	Montreal
Jeremy	M	New York

Gender

Male 1 0 0 1

Female 0 1 1 0

City

Vancouver 1 0 0 0

New York 0 1 0 1

Montreal 0 0 1 0

Figure 1: Bitmap indexing example

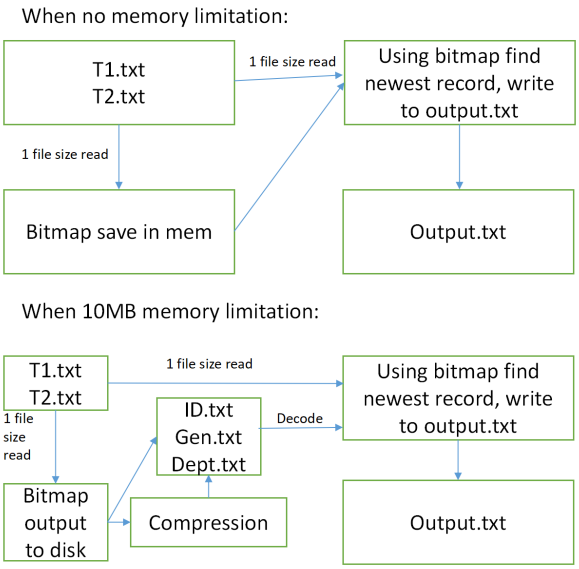

Figure 2: Bitmap indexing flow chart

2.1 Bitmap indexing merge components

Our paper focused on five key components of the bitmap indexing merge, which were the ID bitmap, the Gender bitmap, the department bitmap, the encode-decode and the merge function. Figure.2 shows the flow chart of basic procedures.

2.2 Bitmap for ID

The bitmap for ID substring(0,8) used in our implementation comes from the following form in Figure.3. When reading the original tuples, it will refill 10MB memory recursively and output .txt file of the bitmap like the format shown. Here '0' is not shown, '1' is shown. And by traverse the bitmap.txt using for-loop, we can extract duplicate lines and do further operation to eliminate the duplicates. Also, for uncompressed bitmap, it can ignore unused '0000' like Figure.3 shown in order to save much disk I/O. This is also a kind of no-cost loss-less compression. The algorithm to detect duplicate could be used for bitmap ID shown in Figure.4:

Original tuples

```
542982212014-10-18Jodie Purchase          0007147949197 Corbin VA 22446 South
542982212014-10-18Jodie Purchase          0007147949197 Corbin VA 22446 South
542982212014-10-18Jodie Purchase          0007147949197 Corbin VA 22446 South
542982212014-10-18Jodie Purchase          0007147949197 Corbin VA 22446 South
542982212014-10-20Jodie Purchase          0007147949197 Corbin VA 22446 South
542982222014-10-20Jodie Purchase          0007147949197 Corbin VA 22446 South
542982232014-10-18Jodie Purchase          1007147949197 Corbin VA 22446 South
542982232014-10-18Jodie Purchase          0007147949197 Corbin VA 22446 South
542982242014-10-31Jodie Purchase          1007147949197 Corbin VA 22446 South
```

ID	Bitmap	Index	.txt bitmap
54298221	111110000	1-5 lines	11111**0000**
54298222	000001000	6 line	000001**000**
54298223	000000110	7-8 lines	00000011**0**
54298224	000000001	9 line	000000001

Figure 3: Bitmap ID with lossless compression

```
Input: ID(0,8) with bitmap '00001'
For i < size(substring bitmap '00001'){
  if first char is '1'{
     add i to duplicate list  //i indicate the line number
     }
}
For element:duplicate list{
   Read line at original file.
   if list size>2{ compare each elements date, update most recent one}
}
```

Figure 4: Duplicate detect and update

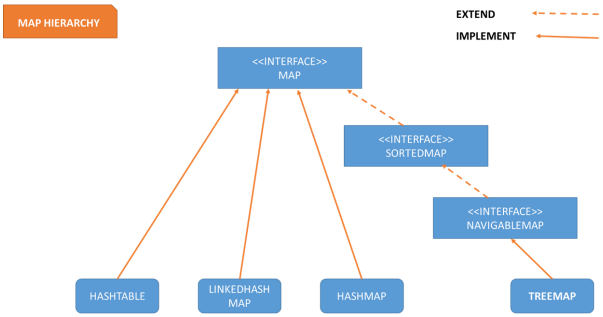

Figure 5: Sorting with ID by tree-map

2.3 Bitmap for gender

The bitmap for gender is one bit. It is marked as either 1 or 0 to represent male and female. Look at the original tuples as an example. On the right side of the data-set, there are 9 lines of 13-length-number. The first line is '0007147949197'. The first number of this index is 0, and that is all we consider by now. Same concept applies to all the tuples. Therefore, for the gender bitmap text in this case, the result we get '000000101'.

2.4 Bitmap sorting

In the project, if we have T1.txt and T2.txt two files, the ID map could generated for each of them, the length with ID map should be unique ID number in the file. In order to do that, the data-structure Tree Map in Java has been used. Once duplicated ID put in the Map, it will update at run-time. The Map functions in Java as shown in Figure.5, and the output will expected as increasing-order with ID, the line number is related to the gender bitmap and department bitmap directly.

2.5 Bitmap for department

The bitmap for department ID includes two parts: department name which is represented as a 3-length number and followed with department-representation in bitmap. For instance, '007111111111' indicates the name of the department is"007" and the rest of the String '111111111' responding to its bits position. In order to give you a better understanding, Figure.6 above shows a simplified example to represent bitmap for department as well. In this

Original tuples

```
542982212014-10-18Jodie Purchase          0007147949197 Corbin VA 22446 South
542982212014-10-18Jodie Purchase          0007147949197 Corbin VA 22446 South
542982212014-10-18Jodie Purchase          0007147949197 Corbin VA 22446 South
542982212014-10-18Jodie Purchase          0007147949197 Corbin VA 22446 South
542982212014-10-20Jodie Purchase          0007147949197 Corbin VA 22446 South
542982222014-10-20Jodie Purchase          0007147949197 Corbin VA 22446 South
542982232014-10-18Jodie Purchase          1007147949197 Corbin VA 22446 South
542982232014-10-18Jodie Purchase          0007147949197 Corbin VA 22446 South
542982242014-10-31Jodie Purchase          1007147949197 Corbin VA 22446 South
```

Dept	Bitmap	Index
007	111111111	1-9 lines

Figure 6: Department bitmap format

example, we have total of 9 tuples. Look at the number on the right. The first index represents gender, it already discussed in last section. Let's ignore it for now. Starting with the second number of those string, the next three numbers indicates their names of department. In this case, all our department names are "007". The rest of the string present bitmap positions for department. Therefore, for the department bitmap text in this case, we get 007111111111.

2.6 Compression

For the compression part, we first try find some better approach than 'Run-Length Encoding', but most of algorithms target the bitmap compression for database quick

query response but not for out project sorting purpose. Therefore, we use DB5.pdf last pages algorithm to encode the bitmap as shown in Figure.7. In the real cases, if the line is many ”000000000...1.000..1” it will decrease the size of the bitmap dramatically, in our Lab2 this shows on ID filed. But for gender its many random ”01011010101...” combination, using the RLE algorithm will sometime increase the size of original code because there not continuing ”00000...” between ”1”. For department, when the number of department is not that huge, it will generating similar thing like ID bitmap, which is compressed to reasonable decreased size. But for some cases which the bitmap is continuing ”11111...1” with no ”0” here, the size of RLE-encoded bitmap will increased as shown in Figure.12, from format “(j-1) 1‘s+0+the actual run length (j) in binary” we could get some additional ”0” will be added, that is the shortage of this encoding, only suitable for bitmap with a sequence of 0’s followed by a 1.

```
char [] bitmapArray = tobeEncode.toCharArray();
ArrayList<Integer> counterArraylist = new ArrayList<>();

int count = 0;
for(int i=0; i < bitmapArray.length; ++i){
    char c = bitmapArray[i];
    if(c == '0')
        count++;
    else{
        counterArraylist.add(count);
        count = 0;
    }
}

String compressLine = "";
for (int num:counterArraylist) {
    String part1 = "";
    String part2 = new String(Integer.toBinaryString(num));
    int part2Len = part2.length();
    for(int i = 0; i<part2Len-1; ++i){
        part1 = new String (part1 + "1");
    }
    part1 = new String(part1 + "0");
    compressLine = new String(compressLine + part1 + part2);
}
```

Figure 7: Compression algorithm

2.7 Merge by bitmap

Another very important part is to split bitmap and merge bitmap pieces, the challenge thing for this is memory only 10MB. For example, if the size of input file is 10000 tuples with 10000 none-duplicate ID’s, the bitmap length for each ID should be 10000. Save this to bitSet in java will cost memory, and it will generate warning of heap overload. In order to avoid heap overload, the bitmap will be generated as shown below in Figure.8.

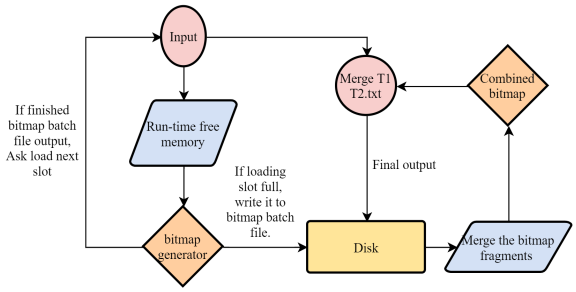

Figure 8: Bitmap fragments and merge

First, the bitmap can’t always save in 10MB memory, therefore a sorted bitmap batch file will generated according to the run time free memory. Once get many sorted batch file of bitmaps for ID, we can merge them together, than finally can use combinedbitmap.txt to do the T1/T2 two txt files merge as shown in Figure.9.

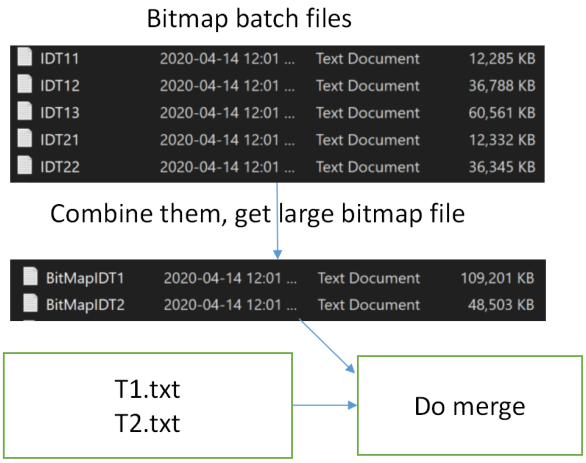

Figure 9: Merge the files

Second, for the input T1/T2.txt merge, the bitmap com-

bined file will be used for indicating the duplicated lines, by reading sorted bitmap file 1st line at same time, compare the ID to find which is larger, which is smaller, than we could get lowest to highest ID order in output file. After that, for each bitmap line it recorded the location and could be used to detect duplicates, by extract original tuples from input file as shown in Figure.10, we could update the each output line as required.

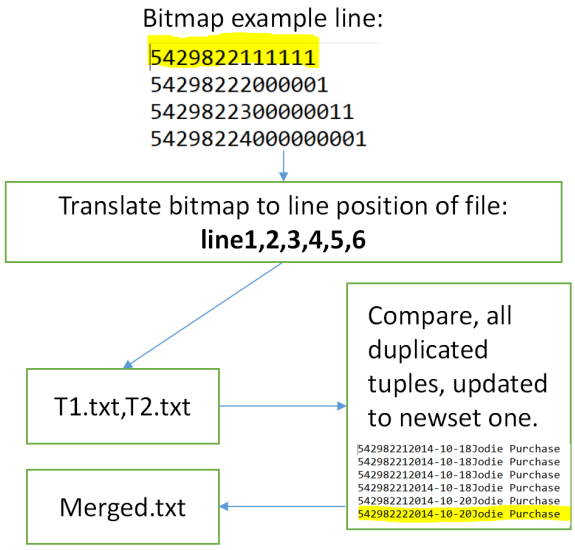

Figure 10: Merge by bitmap

3 Experimental Results

In this section, all results are post here. The test cases are generated with Lab1 TPMMS and Lab2 bitmap indexing separately.

3.1 Bitmap indexing time

All of the time results are indicated in Figure.11, we could find that when bitmap become very long, for 50,000 tuples it could be each bitmap has length 50,000 if no duplicates, which means exponential time consuming here.

Input tuples	ID time	Dept/Gen time	Encoded time	Merge time	Total time	Total I/O
20000	7.317 s	1.31 s	7.869 s	32.814 s	41.441 s	104310012 bytes
25000	8.3 s	1.87 s	9.678 s	47.785 s	57.081 s	164287197 bytes
50000	21.449 s	3.078 s	24.589 s	154.881 s	185.472 s	660660652 bytes
100000	59.767 s	17.302 s	77.41 s	703.536 s	780.605 s	2521020312 bytes

Figure 11: Bitmap time

3.2 Bitmap indexing sizes

The size results are indicated in Figure.12, when bitmap encoded as RLF algorithm, it could save tons of spaces and disk I/O on ID bitmap. For 50,000 tuples it compressed from 2.4GB to 4MB, significantly improved the space cost.

Input tuples	ID size	Dept size	Gender size	Encoded ID	Encoded Dept	Encoded Gen
20000	100190000 bytes	20008 bytes	20004 bytes	694472 bytes	40010 bytes	40004 bytes
25000	161487263 bytes	25008 bytes	25004 bytes	880884 bytes	50012 bytes	50008 bytes
50000	650360640 bytes	50008 bytes	50004 bytes	1868828 bytes	100004 bytes	100010 bytes
100000	2500420300 bytes	100008 bytes	100004 bytes	3937408 bytes	200010 bytes	200004 bytes

Figure 12: Bitmap size

3.3 Bitmap vs TPMMS

All of the time and I/O results are indicated in Figure.13, obviously, due to the cost for establish bitmap to the disk, there are too many extra space and time costs related to save and read uncompressed bitmap. TPMMS shows much better performance on merge phase, bitmap indexing has very good performance on gender/dept indexing, but for ID is not as good as TPMMS.

Input tuples	Bitmap I/O	Bitmap time	TPMMS I/O	TPMMS time
20000	104310012 bytes	41.441 s	6120000 bytes	0.51 s
25000	164287197 bytes	57.081 s	7650000 bytes	0.56 s
50000	660660652 bytes	185.472 s	15300000 bytes	0.79 s
100000	2521020312 bytes	780.605 s	30600000 bytes	1.14 s

Figure 13: Bitmap size

4 Conclusion

In this project, we first implemented bitmap indexing method as textbook indicated[1], got a reasonable great result. After that, we did several experiments on our implemented algorithm, find more results related to input size and bitmap size, and it indicated the most critical challenge is memory limitation.

A Work distribution

Using eclipse to run "Lab2/bitmap/bitmapmain.java" file in order to reproduce the results.

Tianlin Yang: All code implementation, testing, debug, record results, and report writing.

Peiqi Wang: Report writing.

Zelan Xiao: Code implementation, testing, debug.

References

[1] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, USA, 2 edition, 2008. 6

[2] Daniel Lemire, Owen Kaser, and Kamel Aouiche. Sorting improves word-aligned bitmap indexes. *Data Knowledge Engineering*, 69(1):3–28, Jan 2010. 1

[3] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi-Yan-Kai. Roaring bitmaps: Implementation of an optimized software library. *Software: Practice and Experience*, 48(4):867–895, Jan 2018. 1