Mobile Data Compression using e-cloud

A dissertation submitted in the partial fulfillment for the award of Degree of

Master of Technology

In

Software Technology

by

Jatin (Roll no. 2K11/SWT/08)

Under the Essential guidance of

Mr. Manoj Kumar

Associate Professor

DEPARTMENT OF SOFTWARE ENGINEERING

DELHI TECHNOLOGICAL UNIVERSITY

BAWANA ROAD, DELHI

2011-2014

DECLARATION

I hereby want to declare that the thesis entitled “**Mobile Data Compression using e-cloud”** which is being submitted to the **Delhi Technological University**, in partial fulfillment of the requirements for the award of degree in **Master of Technology in Software Technology** is an authentic work carried out by me. The material contained in this thesis has not been submitted to any institution or university for the award of any degree.

Jatin

Department of Software Engineering

Delhi Technological University,

Delhi.

CERTIFICATE

DELHI TECHNOLOGICAL UNIVERSITY

BAWANA ROAD, DELHI-110042

Date:

This is to certify that the thesis entitled **“Mobile Data Compression using e-cloud”** submitted by **Jatin (Roll Number: 2K11/SWT/08),** in partial fulfillment of the requirements for the award of degree of Master of Technology in Software Technology, is an authentic work carried out by him under my guidance. The content embodied in this thesis has not been submitted by him earlier to any institution or organization for any degree or diploma to the best of my knowledge and belief.

Mr. Manoj Kumar ,

Associate Professor, Department of Software Engineering,

Delhi Technological University, Delhi-110042

ACKNOWLEDGEMENT

I would like to take this opportunity to express my appreciation and gratitude to all those who have helped me directly or indirectly towards the successful completion of this work.

Firstly, I would like to express my sincere gratitude to my guide **Mr. Manoj Kumar**, **Associate Professor, Department of Software Engineering**, **Delhi Technological University, Delhi** whose benevolent guidance, encouragement, constant support and valuable inputs were always there for me throughout the course of my work. Without her continuous support and interest, this thesis would not have been the same as presented here.

In addition, I would like to extend my thanks to the entire staff in the Department of Software Engineering, DTU for their help during my course of work.

Jatin

2K 11/SWT/ 08

ABSTRACT

Data processing has been existing as a field since the origin of computer science. However, the interest for data processing increased recently due to the present extension of Internet communication, and to the fact that nearly all texts produced today are stored on, or transmitted through a computer medium at least once during their lifetime. In this context, the processing of large, unrestricted texts written in various languages usually requires basic knowledge about words of these languages. These basic data are stored into large data sets called lexicons or electronic dictionaries, in such a form that they can be exploited by computer applications like spelling checkers, spelling advisers, typesetters, indexers, compressors, speech synthesizers and others. The use of large-coverage lexicons for data processing has decisive advantages: Precision and accuracy: the lexicon contains all the words that were explicitly included and only them, which is not the case with recognizers like spell. Predictability: the behavior of a lexicon-based application can be deduced from the explicit list of words in the lexicon. In this context, the storage and lookup of large-coverage dictionaries can be costly. Therefore, time and space efficiency is crucial issue.

In mobile most of the words are repeating it again and again and there are lot of compression technique. It has been observed that LZ trie is best of them if the data is similar in most of the words or sentences.

Trie data structure is a natural choice when we think about storing and searching over sets of strings or words. In the contemporary usage of the term, a trie for a set of words is a tree in which each transition represents one symbol (or a letter in a word), and nodes represent a word or a part of a word that is spelled by traversal from the root to the given node. The identical

prefixes of different words are therefore represented with the same node and space is saved where identical prefixes abound in a set of words - a situation likely to occur with natural language data. The access speed is high, successful look up is performed in time proportional to the length of word since it takes only as many comparisons as there are symbols in the word. The unsuccessful search is stopped as soon as there is no letter in the trie that continues the word at a given point, so it is even faster.

With the above technique we can compress the static data but data is changing continuously. In order to make it dynamic, we will compress the static data and will keep the separate database for the modify/delete/add entries and then send the compress data along with the database that stores the modified data to e-cloud in order to make it faster.

At eloud, decompression take place for compressed data and appended/modified the list as per the modification and them compression take place and send it to mobile.

Based on the results, it is concluded that LZ Trie is most suitable compression technique in terms of memory saving. It is having the time constraint for compressing the data if the data is very large which can be overcome by doing all these operations at e-cloud.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

INTRODUCTION

Now a day’s data are increasing at a very fast rate and we need a large amount of storage to save it. However, there are many systems like mobile phones where we have the memory constraints. Therefore, we need to compress the data so that more data can be saved in the available memory. In computer science data compression, source coding or bit-rate reduction involves encoding information using fewer bits than the original representation.

Data Compression

The process of reducing the size of a data file is popularly referred to as data compression, although it’s formal name is source coding (coding done at the source of the data before it is stored or transmitted.

Compression can be either lossy or lossless. Lossless compression reduces bits by identifying and eliminating statistical redundancy. No information is lost in lossless compression. Lossy compression reduces bits by identifying unnecessary information and removing it.

Compression is useful because it helps reduce resource usage, such as data storage space or transmission capacity. Because compressed data must be decompressed to use, this extra processing imposes computational or other costs through decompression. Data compression is subject to a space–time complexity trade-off.

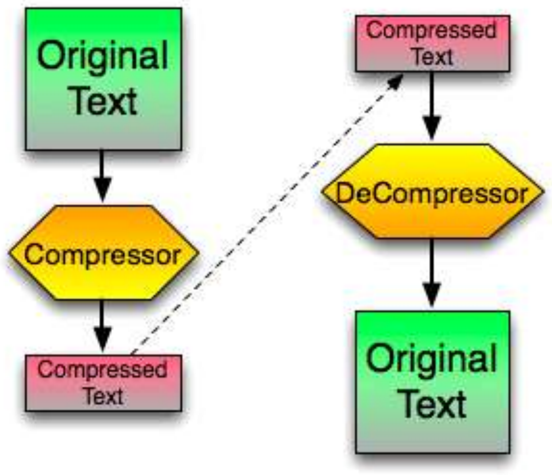


Figure1: Data compression and Decompression

There are many applications that involve storing and accessing finite static sets of strings. These may be sets of simple strings or sets of annotated strings. A simple string lookup is a part of applications such as spelling checkers and word games, while annotated strings are used as keys for accessing the data associated with the strings in dictionaries and database indexes. We shall call a set of simple strings a lexicon, and a set of strings with associated data a dictionary. In addition, we'll use the term enumerated string set for a set of strings where each string has a number, preferably unique, attached to, or derived from it. This number is used to access the data associated with the string. A natural way to enumerate strings in a set of strings is to use their positions in the alphabetically ordered set. A dictionary can be a natural language dictionary (language-to-language or linguistic) or any other translation table that associates some data to a key, like IP number/name database. The dictionary can be implemented as a simple set of compound strings where each string consists of a key and its associated data.

Alternatively, dictionary can be organized as two sets of strings with the keys in the first set and the data in the second. Then, keys need to be enumerated in such a way that the number associated with each key can be used to access the appropriate entry in the data set. Enumerated strings are also needed to access the database indexes.

A trie is a tree where paths from the root to leaves correspond to input words. A trie was first introduced over four decades ago as a means for quick search in a small set of keys As the need for storing larger key sets had developed, improvements were made in order to reduce the trie space requirements. The compression was based mainly on exploiting the sparseness immanent to complete tries for big key sets. A lot of research effort has been put into it and various levels of compression are achieved for both dynamic and static tries If a trie is built with only one character per transition, then this structure, called a character trie or a digital search tree, is a case of a deterministic finite automaton (DFA). An example of such a trie for eight strings is presented in below Figure No 2 . The layout is that of a DFA implemented as a Mealy automaton with root starting state, symbol labeled transitions and eight accepting transitions. Although a standard graphical representation of a DFA would involve accepting states (as in Moore automata) instead of accepting transitions (as in Mealy automata), this layout is more in agreement with the logic of implementation that will be presented later in the text.

The strings displayed or stored in the mobile (text strings and the soft keys strings) for the various languages are stored in the separate text files for each language. In the low end phones these text files are stored as a part of the binary. So in order to display any string, we have to refer to the text file, according to the set phone language and pass the ID of the string and accordingly the value of the string is returned.

In the new approach, rather than storing the text files as such, the files are first compressed using the LZ-Trie String compression algorithm and these compressed files are then stored in the binary. While displaying the string, the id is first mapped to the id of the compressed file which is then used for accessing the string. As the strings are static, thus the approach is called as Static String Compression Algorithm.

By compressing the string files the space used for storing the text files is considerably reduced.

For storing the contacts in compressed form, which is dynamic we need to alter the compress data on regular basis which is based on some algorithm (regular interval, after modification of x entries, low CPU utilization, night etc). In this case we compress the data after some time, then maintain the separate database for the updated entries, and then again compress the data along with the modified one. In this way we are compression the dynamic data using static string compression.

1.1 Motivation of the Work

To assure, more and more features/ solutions can be provided at a low memory segment phone. There is a need to compress the data and find out the efficient compression technique.

The fundamental principles are as follows:

- Find out the efficient compression algorithm
- Trade off between the time and space
- Complexity
- Type of Data

In low segment mobile phones, most of the data are in the form of strings like phonebook data, display strings. Some of them are static while some are dynamic. We need to compress this data so that more feature support can be provided which is very important in competitive market

1.2 Goals & Major Thesis Contributions

The thesis focuses majorly on the following topics and areas:

- To analyze the various lossless compression techniques for the data in the form of strings
- Development of LZ Trie Algorithm

- Use the Algorithm for compression of dynamic data after doing some modification
- Use the e-cloud to make it compression and decompression faster
- Use the mobile phone as a client for the compression and decompression and e-cloud as a server
- Comparison of different compression algorithm

1.3 Organization of Thesis

The rest of the thesis is organized as follows:

***Chapter* 2** discusses the previous work done in the field of data compression techniques. This includes the extensive study of various compression and decompression techniques that have been proposed in the literature so far.

Chapter 3 is about research methodology used herein. It describes the objectives of this research and also the techniques used for the same. It tells about the design and attributes of the compression and decompression of LZ Trie. We also analyzed how we can use the LZ Trie algorithm for the dynamic data using e-cloud.

Chapter 4 In this chapter we give our final concluding remarks. Also we mention the limitations of this study and what are the plans for any future study.

CHAPTER 2

Literature Survey

This chapter discusses various most widely used compression techniques. The main objective of data compression is to increase the storage with the time/space tradeoff with minimal and no data loss. This chapter also discusses about the introduction of LZ Trie Algorithm.

Introduction:-

Compressions are of 2 types:-

<u>Lossy</u>

In information technology, "lossy" compression is the class of data encoding methods that uses inexact approximations (or partial data discarding) for representing the content that has been encoded. Such compression techniques are used to reduce the amount of data that would otherwise be needed to store, handle, and/or transmit the represented content.

Lossy compression is most commonly used to compress multimedia data (audio, video, and still images), especially in applications such as streaming media and internet telephony.

<u>Loseless</u>

Lossless data compression is a class of data compression algorithms that allows the original data to be perfectly reconstructed from the compressed data

Lossless compression is used in cases where it is important that the original and the decompressed data be identical, or where deviations from the original data could be deleterious. Typical examples are executable programs, text documents, and source code. Some image file formats, like PNG or GIF, use only lossless compression, while others like TIFF and MNG may use either lossless or lossy methods

The main objective of this chapter is to introduce two important lossless compression algorithms: Huffman Coding and Lempel-Ziv Coding. A Huffman encoder takes a block of input characters with fixed length and produces a block of output bits of variable length. It is a fixed-to-variable length code. Lempel-Ziv, on the other hand, is a variable-to-fixed length code. The design of the Huffman code is optimal (for a fixed blocklength) assuming that the source statistics are known a priori. The Lempel-Ziv code is not designed for any particular source but for a large class of sources. Surprisingly, for any fixed stationary and ergodic source, the Lempel-Ziv algorithm performs just as well as if it was designed for that source. Mainly for this reason, the Lempel-Ziv code is the most widely used technique for lossless file compression.

2.1 Huffman Coding

Huffman coding is based on the frequency of occurance of a data item (pixel in images). The principle is to use a lower number of bits to encode the data that occurs more frequently. Codes are stored in a Code Book which may be constructed for each image or a set of images. In all cases the code book plus encoded data must be transmitted to enable decoding.

The Huffman algorithm is now briefly summarised:

A bottom-up approach

1. Initialization: Put all nodes in an OPEN list, keep it sorted at all times (e.g., ABCDE).

2. Repeat until the OPEN list has only one node left:

(a) From OPEN pick two nodes having the lowest frequencies/probabilities, create a parent node of them.

(b) Assign the sum of the children's frequencies/probabilities to the parent node and insert it into OPEN.

(c) Assign code 0, 1 to the two branches of the tree, and delete the children from OPEN.

Table1: Bit required for expressing symbol in Huffman coding

Symbol	**Count**	**log(1/p)**	**Code**	**Subtotal (# of bits)**
A	15	1.38	0	15
B	7	2.48	100	21
C	6	2.70	101	18
D	6	2.70	110	18
E	5	2.96	111	15

TOTAL (# of bits): 87

The following points are worth noting about the above algorithm:

Decoding for the above two algorithms is trivial as long as the coding table (the statistics) is sent before the data. (There is a bit overhead for sending this, negligible if the data file is big.)

Unique Prefix Property: no code is a prefix to any other code (all symbols are at the leaf nodes) -> great for decoder, unambiguous.

If prior statistics are available and accurate, then Huffman coding is very good.

In the above example:

Number of bits needed for Huffman Coding is: 87 / 39 = 2.23

In computer science and information theory, Huffman coding is an entropy encoding algorithm used for lossless data compression. The term refers to the use of a variable-length code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the estimated probability of occurrence for each possible value of the source symbol.

2.2 LZW

Lempel–Ziv–Welch (LZW) is a universal lossless data compression algorithm created by Abraham Lempel, Jacob Ziv, and Terry Welch. It was published by Welch in 1984 as an improved implementation of the LZ78 algorithm published by Lempel and Ziv in 1978. The algorithm is simple to implement, and has the potential for very high throughput in hardware implementations.[1] It was the algorithm of the widely used Unix file compression utility compress, and is used in the GIF image format.

2.2.1 Encoding

A high level view of the encoding algorithm is shown here:

- Initialize the dictionary to contain all strings of length one.
- Find the longest string W in the dictionary that matches the current input.
- Emit the dictionary index for W to output and remove W from the input.
- Add W followed by the next symbol in the input to the dictionary.
- Go to Step 2.

```
w = NIL;
while ( read a character k )
{
    if wk exists in the dictionary
        w = wk;
    else
        add wk to the dictionary;
8  output the code for w;
9  w = k;
10  }
```

Input string is "^WED^WE^WEE^WEB^WET".

Table2: Compression sequence using LZW

w	**k**	**output**	**index**	**symbol**
NIL	^			
^	W	^	256	^W
W	E	W	257	WE
E	D	E	258	ED
D	^	D	259	D^
^	W			
^W	E	256	260	^WE
E	^	E	261	E^
^	W			
^W	E			
^WE	E	260	262	^WEE
E	^			
E^	W	261	263	E^W

W	E			
WE	B	257	264	WEB
B	^	B	265	B^
^	W			
^W	E			
^WE	T	260	266	^WET
T	EOF	T		

2.2.2 Decoding

The decoding algorithm works by reading a value from the encoded input and outputting the corresponding string from the initialized dictionary. At the same time it obtains the next value from the input, and adds to the dictionary the concatenation of the string just output and the first character of the string obtained by decoding the next input value, or the first character of the string just output if the next value can not be decoded (If the next value is unknown to the decoder, then it has just been added, and so its first character must be the same as the first character of the string just output). The decoder then proceeds to the next input value (which was already read in as the "next value" in the previous pass) and repeats the process until there is no more input, at which point the final input value is decoded without any more additions to the dictionary.

```
1  read a character k;

2  output k;

 3 w = k;

 4 while ( read a character k )

  /* k could be a character or a code. */
```

```
5  entry = dictionary entry for k;

6  output entry;

7  add w + entry[0] to dictionary;

8  w = entry;

9  }
```

Input string is "^WED<256>E<260><261><257>B<260>T".

Table3 : Decompression sequence using LZW

w	**k**	**output**	**index**	**Symbol**
	^	^		
^	W	W	256	^W
W	E	E	257	WE
E	D	D	258	ED
D	<256>	^W	259	D^
<256>	E	E	260	^WE
E	<260>	^WE	261	E^
<260>	<261>	E^	262	^WEE
<261>	<257>	WE	263	E^W
<257>	B	B	264	WEB
B	<260>	^WE	265	B^
<260>	T	T	266	^WET

CHAPTER 3

Research Methodology

In the new approach used for compression, first each of these text files are first compressed and then these compressed files are stored in the binary. These compressed files are then used to refer for displaying of the strings.

The approach used can be broadly divided into 2 categories:

- Compression Algorithm
- Decoding Procedure

3.1 Encoding Algorithm

For compression algorithm comprises of following steps:-

- Firstly each of the text files are parsed and sorting algorithm is applied to sort the strings.
- Generation of trie from the strings in sorted manner.
- Generation of the linked list corresponding to the generated trie.
- Marking of the repeated subsequence.
- Populating the global variables to be written in the compressed files
- Sorting of the generated array and generation of Id Conversion array
- Writing of the generated data in the file.

3.2 Trie Generation

Trie

A trie is a tree where paths from the root to leaves correspond to input words. A trie was first introduced over four decades ago as a means for quick search in a small set of keys. If a trie is built with only one character per transition, then this structure, called a character trie or a digital search tree, is a case of a deterministic finite automaton (DFA).

Consider the following strings aijaklm, aijaxy, bijbklm, bijbxy, cijcklm, cijcxyz, dijdklm, dijdxyz. From these strings the trie is generated as:-

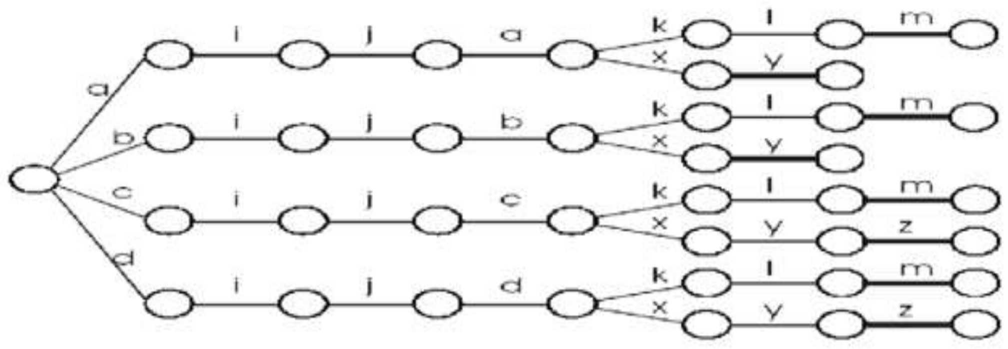


Figure 2. Trie generated corresponding to the 8 strings aijaklm, aijaxy, bijbklm, bijbxy, cijcklm, cijcxyz, dijdklm, dijdxyz.

For simplicity and maintaining only one way pointer from root to other node, while making the trie we insert the string in the sorted manner in the trie. The node is made the sibling node if the node value is greater than the current node. Thus while insertion in the trie the nodes are parsed till we find the node having the value greater than the node value.

3.2.1 Algorithm for Generation of Trie

Data structure used:

A structure treenode having members

Table 4 : Parameters description for generation of Trie

Members	**Description**
word_end	For denoting the end of the string
data	For storing the character value
sibbling_node	It is a pointer node pointing to the sibling node
child_node	Pointer node pointing to the child node

<u>Procedure Trie_Generation</u>

```
1. for each string i
        i. Call InsertUTF8StringInPointerTrie(UnsortPhoneString[i])
2. end
```

<u>Procedure InsertUTF8StringInPointerTrie (Char *str)</u>

```
    1. Initialize CurrentTrieNode = &RootNode
    2. While str != NULL
        a) StrData = Store the character value str[i]
        b) if (CurrentTrieNode == NULL)
            1) Initialize CurrentTrieNode.data=StrData
            2) If (tempnode != NULL)
                CurrentTrieNode.sibbling_node= tempnode
            3) tempnode==NULL
            4) if(str +1 ==NULL)
                CurrentTrieNode.wordend= 1
                5) CurrentTrieNode=&(*CurrentTrieNode)->child_node;
                6) str=str+1
        c) else
                1) if((*CurrentTrieNode)->data==(StrData))
                        tempnode=NULL;
                        if(*(str+1)==0)
                                (*CurrentTrieNode)->word_end=1;
                        CurrentTrieNode=&(*CurrentTrieNode)->child_node;
                        str=str+1
                2) else if((*CurrentTrieNode)->data>(StrData))
                        tempnode=*CurrentTrieNode;
                        *CurrentTrieNode=NULL;
                3) else
                         tempnode=NULL;
                        CurrentTrieNode=&(*CurrentTrieNode)->sibbling_node
        d) end if
    3. End of while loop
```

3.3 Linked List Generation from the Trie

The most compact way representing the trie graph is to represent using it using the linked list where each node having the information regarding

1) Symbol

2) Distance, measured in number of elements, to the next element belonging to the same node.

3) One bit flag indicating whether the word ends with the current symbol

4) One bit flag indicating whether there is a continuation of word past the current symbol.

For example the trie shown in figure 2 can be represented in the form of the linked list. The "empty" ∈ symbol indicates that this field is unused. Nodes are formed by linking the elements within-node offsets. For example, elements 1, 10, 19 and 29 form one node, elements 5 and 8 another, etc. The building elements with empty in-node offset field indicate the last (or the only) symbol in the node.

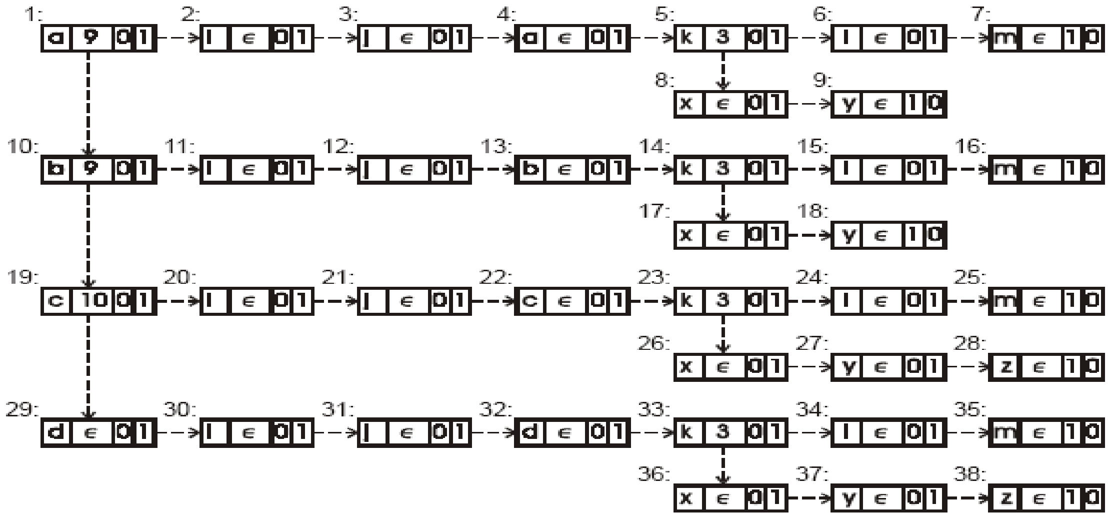


Figure 3. Linked list trie equivalent to that of Figure 2.

3.3.1 Data Structure Used

Each entry in the linked list table contains the node information. It is structure having members containing information about tree node. Its members are:-

Table 5: Parameters description for generation of linked list

S.No	Data Structure Members	Description
1	Data	Contains the node data information, it contains the of tree node data

2	sibbling_node	Have information about distance from the current node to its sibling node
3	sibbling_of	Points to the parent node
4	ifWordEnd	It is the bit specifying that end of the word
5	isLeafNode	It specifies that the node is the leaf node
6	wordendcount	Specifies that the character is a part of how many words. This value is set for the nodes which have sibling nodes and which are sibling of any node.
7	Mergedpointernum	It is used to contain the information regarding merged nodes. It is a pointer in the merged node where each node points to the referring merging node.
8	IsMergeStartNode	It is used to mark the starting of the merged node
9	Numberofmergenode	It is used to store the number of the merged nodes.
10	DataArrayOffset	It is array containing the all the information about the node.

<u>Procedure GenerateTableData(RootNode,SibbIndex)</u>

```
1.      Index++
2.      Table[Index].data=RootNode->data
3.      if(SibbIndex)
        Table[SibbIndex].sibbling_node=Index-SibbIndex;
        Table[Index].sibbling_of=SibbIndex;
4.      SibbIndex=Index;
5.      if(RootNode->child_node)
        GenerateTableData(RootNode->child_node,0);
        else
        Table[SibbIndex].isLeafNode=1
        endif
6.      if(RootNode->word_end)
        Table[SibbIndex].ifWordEnd=1;
        word_end_count++
7.      if(RootNode->sibbling_node)
        Table[SibbIndex].wordendcount=word_end_count;
        word_end_count=0;
        Call GenerateTableData(RootNode->sibbling_node,SibbIndex);
        word_end_count=word_end_count+Table[SibbIndex].wordendcount;
8.      else if(Table[SibbIndex].sibbling_of)
        Table[SibbIndex].wordendcount=word_end_count
```

Marking of the repeated subsequence

3.4 Generation of MDFA

Trie (DFA represented in Figure 2.) can be minimized to produce minimal deterministic finite automaton (MDFA). Deterministic finite automaton stores a finite set of words or a language, and for each language there exists an automaton with the minimal number of states. Minimization is performed by merging the equivalent states in the automaton. A conventional procedure for automata minimization involves building a trie and searching for the equivalent states. For example the trie in figure 2 can be reduced by merging the repeated prefix “lm” present in the strings bijbklm, cijcklm and dijdklm by pointing to its first appearance.

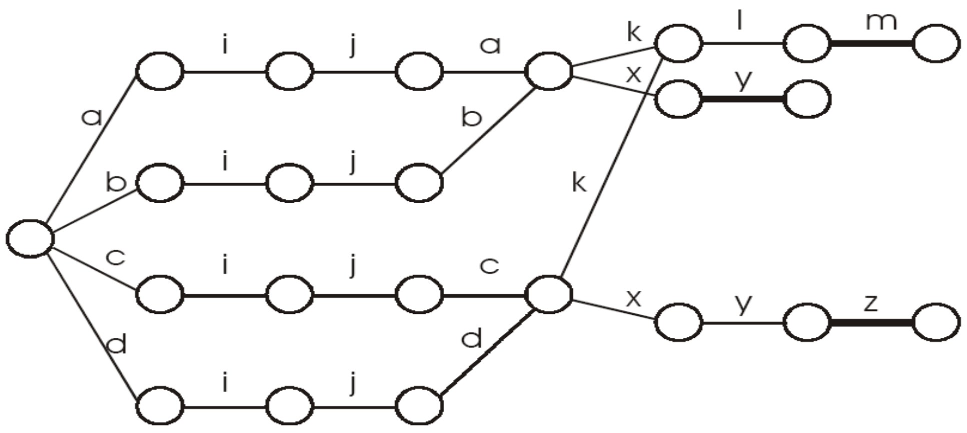


Figure 4. Minimal deterministic finite automaton equivalent to the trie of Figure 2

This tree structure can be represented using the linked list. In order to maintain the same structure of the node as that of the pointer node the same four members as that of the node structure are used to store the pointer node information. In case of the pointer node these contains the information regarding:-

- Length of the repeated subsequence:- This length is used to denote how many elements, when searching the structure, must be checked before returning and continuing from the position of the pointer. Since in case of the MDFA, there is no need to return at the pointer position which means that the pointer is to be replaced the complete branch in the trie. In our case we are specifying it with 0.
- Address of the first occurrence of the repeated subsequence
- 0
- 0

The pointer node is differentiated from the normal node by the last two bits. In case of the pointer node both these two bits have the value 0.

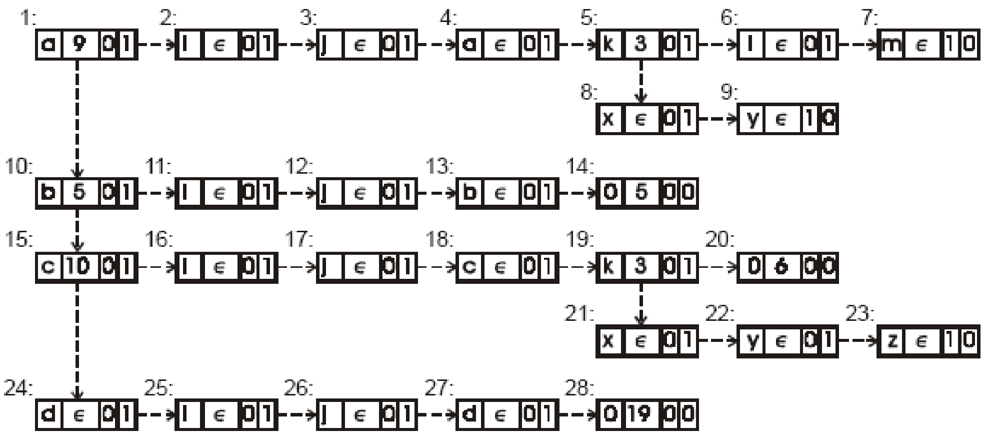


Figure 5. Linked list representation of the Figure 4. Node 14,20,28 are one way pointer node.

3.5 Generation of LZ-Trie

Merging equivalent states in a trie (Figure 2) to produce MDFA (Figure 3) effectively reduces redundancy in a trie by substituting all identical repeated branches with only one. However, this may still leave a number of repeated identical subsections of a trie. These repeated subsections can be replaced with pointers to its first occurrence as demonstrated in Figure 6. If a pointer is smaller in size than the replaced part, then overall size is reduced according to LZ compression paradigm. Hence such compression is called as LZ trie.

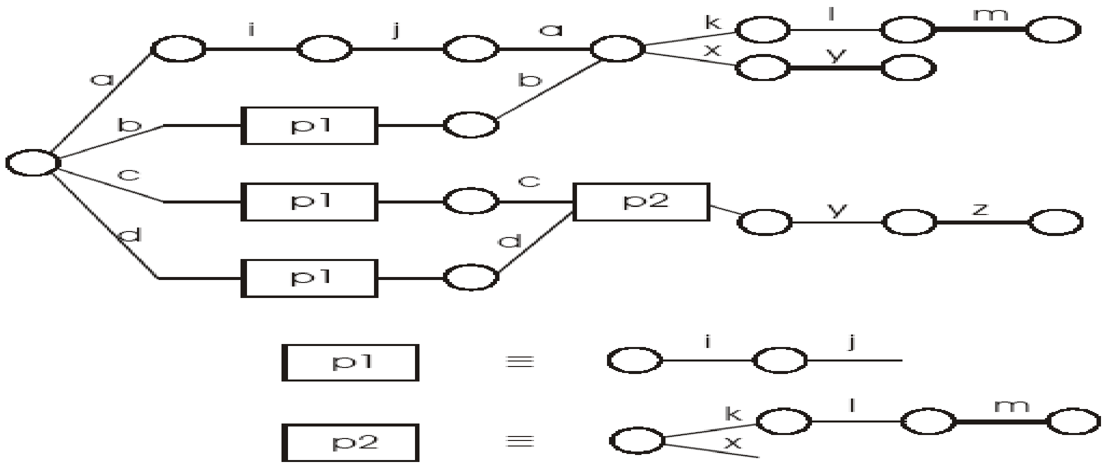
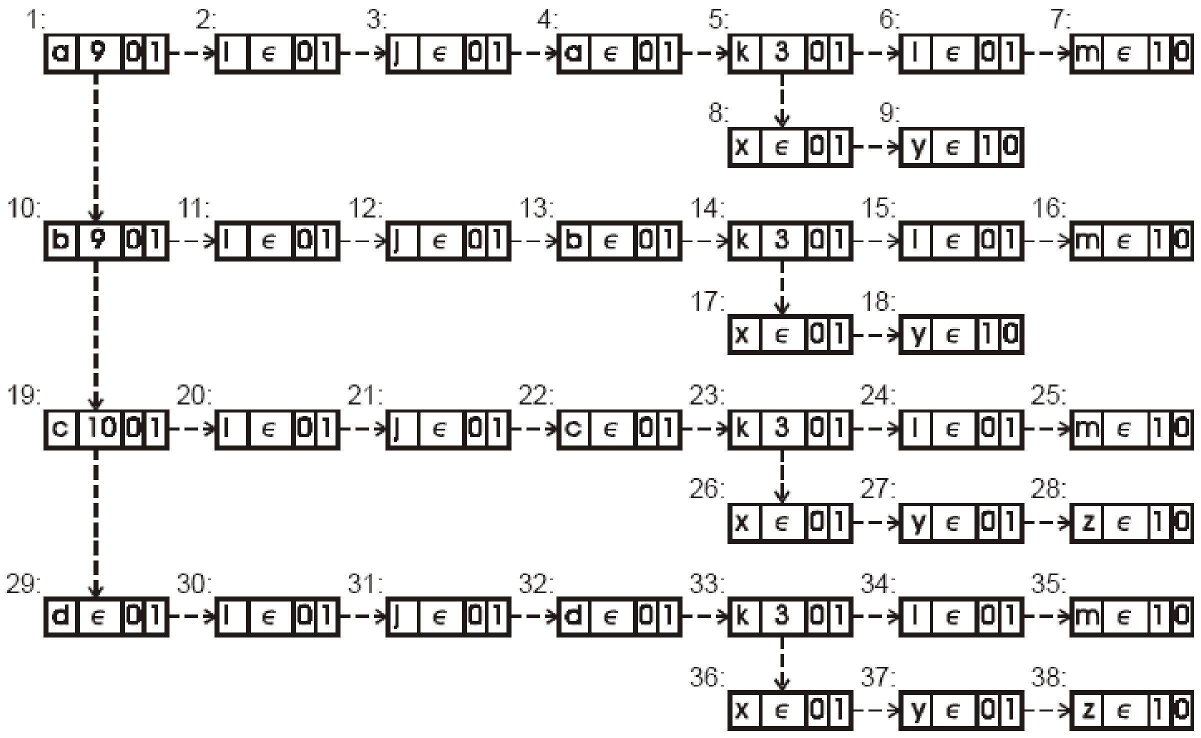


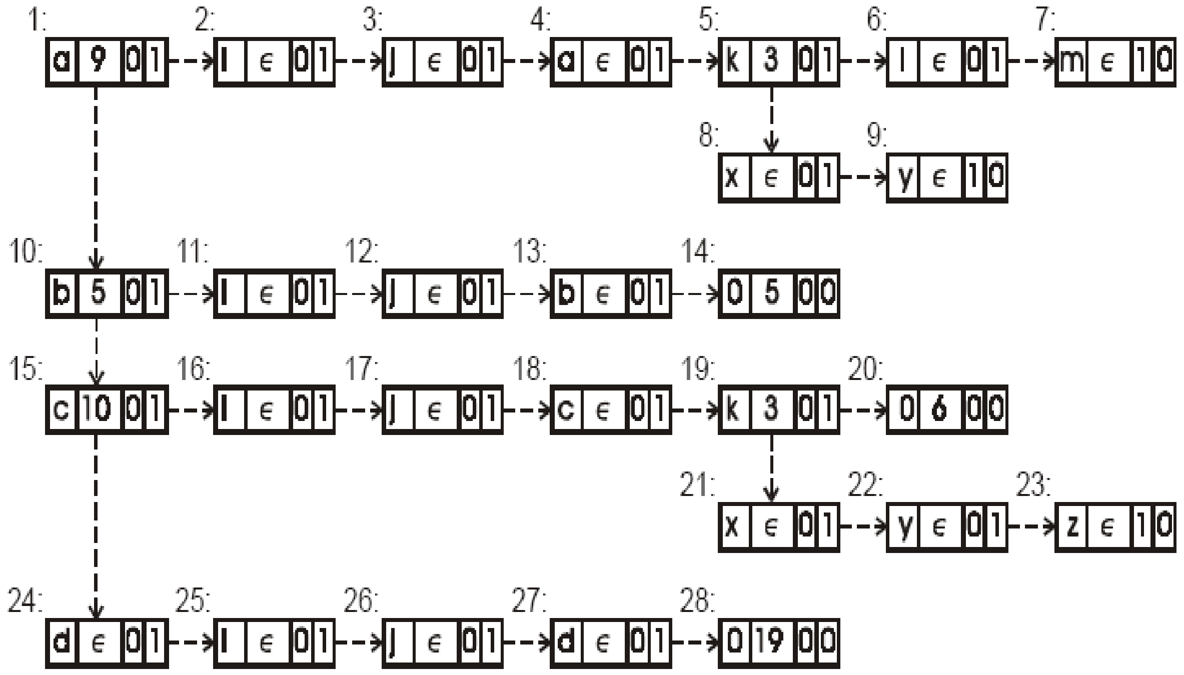
Figure 6. LZ trie equivalent to the trie of Figure 2 and MDFA of Figure 3. p1 and p2 are pointers substituting the repeated parts of MDFA.

Data Structures Used

Table 6 : Parameters for generation of MDFA

	Field 1	**Field 2**	**Field 3**	**Field 4**
For Storing the node	Symbol Code	In-Node Offset	End of word bit flag (0/1)	Continuation of word bit-flag (0/1)
For Storing the pointer	Length of repeated subseq.	Address Of the first Pointer Occurrence	0	0


1: a 9 0 1
2: l ϵ 0 1
3: j ϵ 0 1
4: a ϵ 0 1
5: k 3 0 1
6: l ϵ 0 1
7: m ϵ 1 0
8: x ϵ 0 1
9: y ϵ 1 0
10: b 9 0 1
11: l ϵ 0 1
12: j ϵ 0 1
13: b ϵ 0 1
14: k 3 0 1
15: l ϵ 0 1
16: m ϵ 1 0
17: x ϵ 0 1
18: y ϵ 1 0
19: c 10 0 1
20: l ϵ 0 1
21: j ϵ 0 1
22: c ϵ 0 1
23: k 3 0 1
24: l ϵ 0 1
25: m ϵ 1 0
26: x ϵ 0 1
27: y ϵ 0 1
28: z ϵ 1 0
29: d ϵ 0 1
30: l ϵ 0 1
31: j ϵ 0 1
32: d ϵ 0 1
33: k 3 0 1
34: l ϵ 0 1
35: m ϵ 1 0
36: x ϵ 0 1
37: y ϵ 0 1
38: z ϵ 1 0


1: a 9 0 1
2: l ϵ 0 1
3: j ϵ 0 1
4: a ϵ 0 1
5: k 3 0 1
6: l ϵ 0 1
7: m ϵ 1 0
8: x ϵ 0 1
9: y ϵ 1 0
10: b 5 0 1
11: l ϵ 0 1
12: j ϵ 0 1
13: b ϵ 0 1
14: 0 5 0 0
15: c 10 0 1
16: l ϵ 0 1
17: j ϵ 0 1
18: c ϵ 0 1
19: k 3 0 1
20: 0 6 0 0
21: x ϵ 0 1
22: y ϵ 0 1
23: z ϵ 1 0
24: d ϵ 0 1
25: l ϵ 0 1
26: j ϵ 0 1
27: d ϵ 0 1
28: 0 19 0 0

The linked list structure can be represented for the LZ-trie by updating the information of length of the repeated subsequence which in this case would contain the length of the repeated subsequence rather than 0. These nodes are called as the two way pointer.

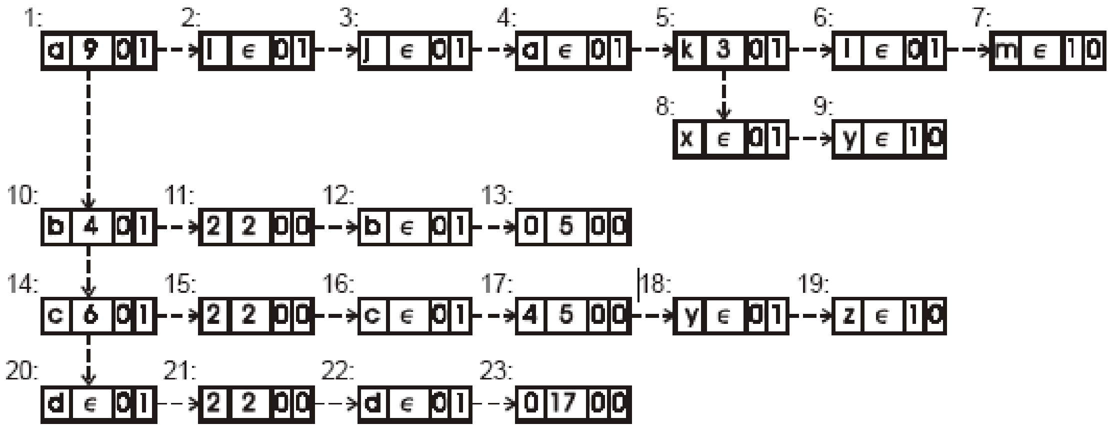


Figure 7. Linked list representation of figure 6 with Elements 13 and 23 are one way pointers, and 11, 15, 17 and 21 are two-way pointers.

However for simplicity and for maintaining the pointer node structure while marking the repeated subsequence following points are to be taken care of:-

- Recursive pointers are not allowed so we must take care for the repeated substring not to overlap with the original one.
- Replacement is not done if, any node in repeated substring is pointing outside the repeated one
- Also replacement is not allowed if any node in repeated substring is pointed by a node earlier than first node of repeated string

<ins>Algorithm</ins>

In our case while finding the repeated subsequence first we start with finding of 15 characters matching and find all such subsequences and mark them. When all the repeated subsequences with 15 characters are found, we decrease the min number of nodes that should be there in the repeated subsequence by 1 and find all such patterns until min number of nodes in the repeated subsequence becomes less than 2.

Procedure CheckForMergeNode (total_nodes)

```
1)  Initialize merge_count, i=0
2)  Initialize min_node_match=15
3)  while(min_node_match>=2)
    a) while (i<=num-2)
          i) if (Table[i]. Mergedpointernum)
       continue;
       ii) initialize j=i+2
       iii) while(j<num-1)
       a) merge_count=for  each node from start node i  to dest node j till num    check
  for the merged nodes after satisfying the merge criteria's.
    b) if(merge_count>= min_node_match)
                i) Table[j].Numberofmergenode =count;
       ii) if( ! Table[i].IsMergeStartNode)
       Table[i].IsMergeStartNode=1;
       iii) for(temp=0;temp< count; temp++)
       Table[j +temp].Mergedpointernum=i+ temp;
       iv) j=j+1
    c) end of while loop
    d) min_node_match =min_node_match-1;
4)  End of while loop
```

Populating of the global variables

The information of the node of the LZ-Trie linked list is represented using the **20 bits** for each node.   In case of the normal node these bits are populated as

- 1 Bit : is used to mark the word ending (SSC_WORD_END_MARKER_BIT_COUNT)
- 8 bits: are used to store the node data information (SSC_DATA_BIT_COUNT)

- 5 bits: are used store the sibling information(SSC_SIBBLING_BIT_COUNT)
- 5 bits: are used store the word end count
  (SSC_WORD_END__NUMBER_BIT_COUNT)

In case of the pointer node these 20 bits are populated as:-

- 1 bit: Is used to store the validating node info
  (SSC_VALIDATION_NODE_BIT_COUNT)
- 5 bit: Contains the information about the number of merged nodes
  (SSC_MERGE_NODE_NUMBER_BIT_COUNT)
- 14 bit: Are used to store the merged nodes sibling info
  (SSC_MERGE_NODE_ADDR_BIT_COUNT)

Data Structure Used:

- Structure SearchSt used for pushing the nodes in the stack having members
  - Node :- Containing the node information
  - CurrentRemainingMergeNodeCount :- Have information regarding the current count of the nodes that are merged.
  - NextLevelMergeNodeCount :- Containing the information regarding the total number of the merged nodes in the merged pointer
- gSibblingOverFlowCount :- It contains the size of the gSibblingOverFlowArray array.
- gSibblingOverFlowArray :- It contains the sibling information if the sibling information is greater than can be contained in 14 bits in case of pointer node and 5 bits in case of normal node.
- gMergeCountOverFlowCount :- It contains the size of the gMergeCountOverFlowArray array.

- gMergeCountOverFlowArray :- It contains the merged count information is greater than 5 bits.
- gWordEndCountOverFlowCount :- It contains the size of gWordEndCountOverFlowArray array.
- gWordEndCountOverFlowArray :- This array contains the information about the word end count if it exceeds the 5 bits.
- DataArray[MAX_DATA_ARRAY_SIZE] :- It contains the information of the offset of node.

Algorithm

Procedure FillPackDataArray()

```
1. Initialize bit_count =20 (i.e.ONE_NODE_BIT_SIZE) and i=1
2. while (Table[i].data != 0)
3. if(Table[i].Mergedpointernum)
    a. Set first bit to 1 to mark the pointer node.
    b. value=Table[i].Numberofmergenode
    c. if(value>=31)  //as 5 bit are used for no of merge node
        i. Fill the mergecountoverflowarray
        ii. value=31
    d. fill the 5 bits for the number of the merge nodes
    e. value = offset of the sibling which is pointing to the current node;
    f. if value>= 16383 // max value with 14 bits
        i. Fill the sibling overflow array
        ii. value = 16383
    g. Fill the 14 bits having info about the sibling node offset which is being pointed
       out
```

```
        h. Check if within the pointer node any node is sibling of any node (case in which
           the node is pointed by other within the merged node
           If exists
                i. Value= Table[Table[i].sibbling_of].DataArrayOffset/20
               ii. **if** (value > 31)  // 5 bits are used
                       1. Fill the sibling overflow array, Set value =31
              iii. Fill 5 bits for the sibling info
        i. i=i+Table[i].Numberofmergenode
    4. else
        a. if(Table[i].ifWordEnd)
        b. Fill 1 bit for word end info
    5. Fill 8 bits containing the node data
    6. If(node is sibling of any node and that node Mergedpointernum=0)
        a. value = Table[Table[i]. sibbling_of ].DataArrayOffset/20
        b. if value>=31   //5 bits are used
                i. fill the sibling over flow array
               ii. value=31
        c. Fill 5 bits containing the parent node
        d. if (Table[i].wordendcount )
                i. value = Table[i].wordendcount
               ii. if value > =31   //5 bits are used
                       1. fill the wordendoverflow array and set value =31
              iii. fill the 5 bits for the word end count
    7. i=i+1
```

Sorting of the Generated Array and Generation of the ID conversion array

The gSibblingOverFlowArray array generated is sorted before being written in the array. This is done in the procedure SortOverFlowArray.

Corresponding to each string the ID Conversion array, IDConversionArray is generated this array is used while decoding procedure to map the SS value of the requested string to the id used to map in the compressed database files. The procedure used to handle this is GenerateConversionArray.

3.6 Procedure GenerateConversionArray

1. Initialize i=0,value=0
2. i=find the occurrence of the string in the table.
3. Value = add all the word end count of the nodes which have the sibling nodes.
4. Value = value +1 // for the word end information
5. Encode the value of the Value in the 2 bytes of IDConversionArray array.

3.7 Writing Generated information in the corresponding files

The information generated about the nodes,

- gSibblingOverFlowCount ,
- gSibblingOverFlowArray ,
- gMergeCountOverFlowCount,
- gMergeCountOverFlowArray,
- gWordEndCountOverFlowCount,
- gWordEndCountOverFlowArray,
- DataArray
- IDConversionArray
- gCharSetArray

is written in the corresponding text and soft key files for each language. It is handled in the procedure **WriteDBFile.**

Steps for Generation of the Compressed files:-

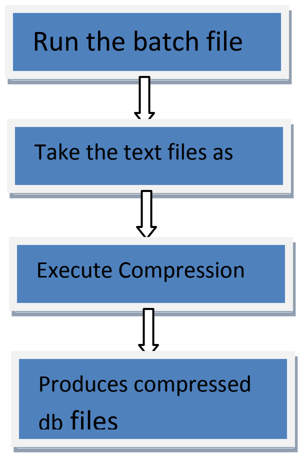


Figure 8. Steps for generation of the compressed files

3.8 Decoding

Once the compressed files are generated these are used for the decoding purposes. The steps followed for decoding are:-

- In the code **get_text (IDS_String),** function is called where IDS is the value of the string to be displayed which in turn is mapped to the SS value. Same as the one followed earlier.

- From the **get_text (IDS_String)** function, function **GetTextFromCompressedDB(SS)** is called which handles all the decoding process. Here **SS** is the value corresponding to the **IDS** defined in the **lkmap.hec** file.

- In the function **GetTextFromCompressedDB**, corresponding to the **gv_TextLanguage** global variable storing the current selected phone language information, global variable **gIntSsc_language_idx** is set.

- The value of the **SS**, is decoded making use of the **IDConversionArray** generated while encoding and stored in the compressed file. This decoded id is further used for searching in the compressed db file.

- Using the decoded ID, the word is searched in the compressed db file. It is handled in the procedure **SearchInCompressedArrayByIndex(ID).** The word corresponding to the ID is searched in the compressed db and is populated in the global decoded array, **gChrSsc_DecodedString**. This function returns the ending position of the array till the word has been added. Using the starting position and the end position, the word is returned from the decoded array.

<u>Algorithm Used for searching the word in the compressed file</u>

Data Structures Used:-

Structure SearchSt used for pushing the nodes in the stack while transversal (same as used in compression). It has elements:-

- Node
- CurrentRemainingMergeNodeCount;
- NextLevelMergeNodeCount;

Structure dbSt is used to map to elements stored in the compressed file. It has elements

- gSibblingOverFlowCount;
- gSibblingOverFlowArray;
- gMergeCountOverFlowCount;
- gMergeCountOverFlowArray;
- gWordEndCountOverFlowCount;
- gWordEndCountOverFlowArray;
- CharSetArray;
- IDConversionArray;
- DataArray

corresponding to each entry in the compressed file.

gChrSsc_DecodedString:- Global array used to store the decoded string.

gIntSsc_DecodedCharCount:- It is the global variable used to store the last entry in the gChrSsc_DecodedString array.

Algorithm:-

Procedure SearchInCompressedArrayByIndex(id)

```
1. Initialize char_count=gIntSsc_DecodedCharCount
2. BitOffset= SSC_ONE_NODE_BIT_SIZE
3. While (id)
    a. check whether the given node is pointer node
            if yes
                a. push the given node information along with total merged
                   count in stack and points the BitOffset to the new valueof
                   the pointed node.
    b. value = decode bits from BitOffset + 1 +1 +8
       (SSC_VALIDATION_NODE_BIT_COUNT +
       SSC_WORD_END_MARKER_BIT_COUNT + SSC_DATA_BIT_COUNT) till
       5 bits (SSC_SIBBLING_BIT_COUNT) to check if the node has any sibling node
    c. if value !=0    //sibling node exists
        i. if value >= 32   //max value for 5 bits
            1. value = Search value in the sibling in the sibling overflow array
        ii. word_end = decode value from offset+1+1+8+5 till 5 bits
        iii. if word_end >= 32
            1. word_end = Search in the wordendoverflow array.
        iv. if id < word_end  // if the information of the word is in sibling node
            1. id=id- word_end
            2. Get siblings of the compressed node and set BitOffset to sibling
               node
    d. charvalue= Decode the node character array offset by decoding 8 bits from
       BitOffset + 1 + 1 (validation+dataend)
```

e. Fetch the character value from the gStSsc_DBInfo[gIntSsc_language_idx] . CharSetArray [charvalue];

f. Check if the part of the string is at the end of the global array

   i. if yes then cut characters and append it to the starting of the global array

   ii. Append the decoded character to the gChrSsc_DecodedString.

g. Check if the stack is not empty

If yes Then pop the entry from the stack and make the BitOffset value points to the value offset information stored in the stack.

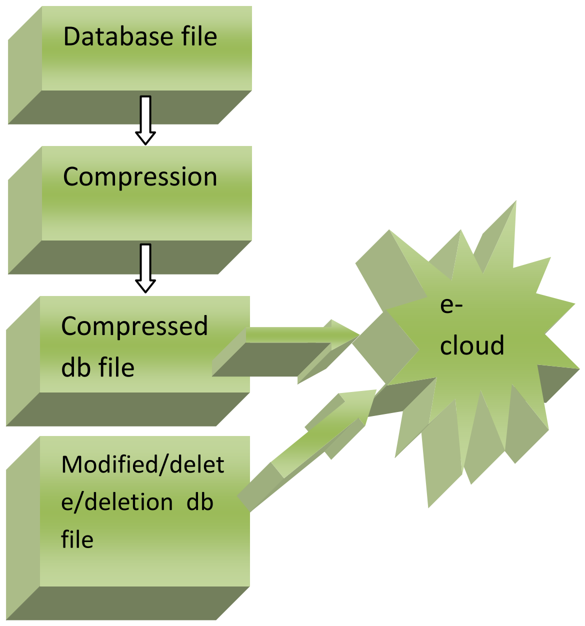


Figure 9: Design Architecture

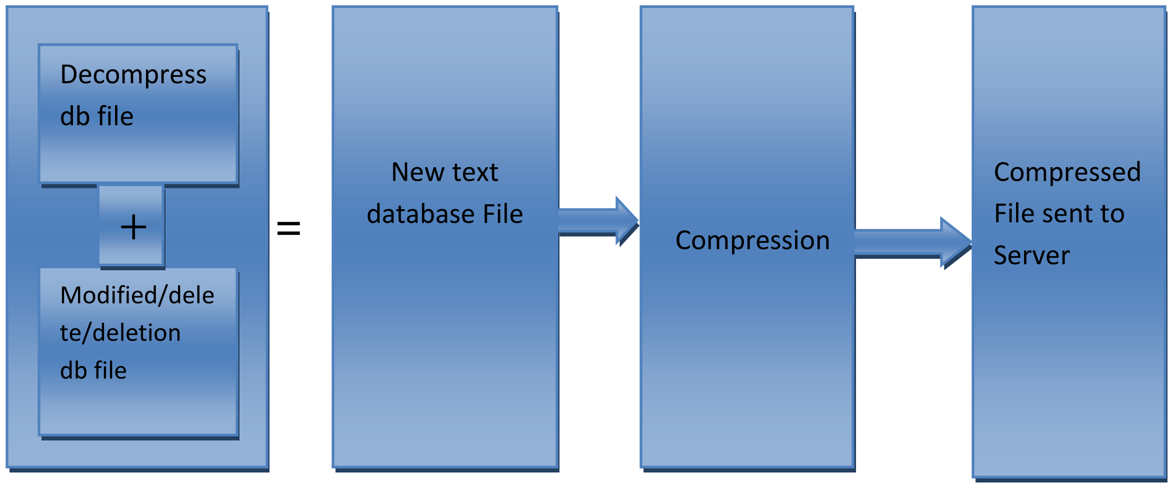

Decompress db file
+
Modified/delete/deletion db file
=
New text database File
Compression
Compressed File sent to Server

Figure 10: e-cloud operation

CHAPTER 4

CONCLUSION AND FUTURE WORK

4.1 Conclusion

The goal of this research is to find out the best compression techniques for the mobile data.

We analyzed the different lompression techniques and observed that modified LZ Trie is the best compression technique specially when the data is static

By making use of the above comression algorithm, the fetching time for decoding increases slightly (however it is still reasonable and does not contribute much of delay).

However the big advantage is that the size of the binary is reduced from **4875.488 Kb** to **4608 Kb** which is quite essential in case of the low end mobile.

Table 7: Compressed data comparison of LZ Trie with Huffman

Lexicon	**Raw Size [KB]**	**Mealy Recognizer[KB]**	**LZ Trie [KB]**	**In % of Mealy recognizer Size**
English	688	270	145	54
French	2418	245	120	49
German	2661	337	189	56
Russian	8911	538	262	49
Random	1027	1688	799	48

The results of this study are that we can made use of above static data algorithm (LZ Trie) for dynamic data with the help of e-cloud where all the compression and decompression takes place.

4.2 Limitations and Future work

This study presented here has its own limitations. These limitations can be summed up as follows:

- It has been observed that decoding time is higher (still it is reasonable) but it can be improved more with the help of some other technique
- The set of quality parameters used by us is limited. This need to be extended to many quality factors (Compression time and complexity of algorithm) that are linked to all the quality factors related to a code.
- Extended the support of static data LZ trie algorithm to dynamic data but can be more optimized .
- The current study has been performed on a particular type of mobile data and this can be conducted on a number of varieties of data and applications. In this way we can infer that which applications are more relevant for this data compression algorithm.
- It will be a humble effort in all our future work to attract the attention of the entire scientific community towards the fact that optimization can actually have a very positive impact in a very effective way.

REFERENCES

[1] Strahil Ristov, “LZ trie and dictionary compression”, in Software: Practice and Experience, June 31, 2008

[2] Strahil Ristov, Eric Laporte, “Ziv Lempel compression of huge natural language data tries using suffix arrays”, combinatorial pattern matching: 10th annual symposium

[3] J. Ziv and A. Lempel, A universal algorithm for sequential data compression, IEEE Transactions on Information Theory, Vol. IT-23, No. 3, 337-343,

[4] S. Ristov, Space saving with compressed trie format, Proceedings of the 17th International Conference on Information Technology Interfaces, eds. D. Kalpic and V. Hljuz Dobric, 269-274, Pula, Jun 1995.

[5] A. Acharya, H. Zhu and K. Shen, Adaptive Algorithms for Cache-efficient Trie Search, ACM and SIAM Workshop on Algorithm Engineering and Experimentation ALENEX 99, Baltimore, Jan. 1999.

[6] YonghuiWu,Stefano Lonardi, Wojciech Szpankowski on Error-Resilient LZW Data Compression, IEEE 2006

[7]D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, 1997.

[8] T. Kowaltowski, C. Lucchesi and J. Stolfi, Finite automata and efficient lexicon implementation, Technical report IC-98-2, University of Campinas, Brazil, 1998.

[9] C. Lucchesi and T. Kowaltowski, Applications of finite automata representing large vocabularies, Software-Practice and Experience, Vol. 23, No. 1, 15-30, 1993.

[10] U. Manber and G. Myers, Suffix arrays: a new method for on-line search, SIAM Journal on Computing, Vol. 22, No. 5, 935-948, 1993.

[11]K. Morimoto, H. Iriguchi and J. Aoe, A method of compressing trie structures, Software-Practice and Experience, Vol. 24, No. 3, 265-288, 1994.

[12]T. D. M. Purdin, Compressing tries for storing dictionaries, Proceedings of the 1990 Symposium on Applied Computing, Fayetteville, Apr. 1990.

[13]D. Revuz, Dictionnaires et lexiques: Méthodes et algorithmes, Ph.D. thesis, CERIL, Université Paris 7, 1991.

[14] S. Ristov, Space saving with compressed trie format, Proceedings of the 17th International Conference on Information Technology Interfaces, eds. D. Kalpic and V. Hljuz Dobric, 269-274, Pula, Jun 1995.

[15] S. Ristov, D. Boras and T. Lauc, LZ compression of static linked list tries, Journal of Computing and Information Technology, Vol. 5, No. 3, 199-204, Zagreb, 1997.

[16]M. Silberztein, INTEX: a corpus processing system, Proceedings of COLING-94, Kyoto, 1994.lexique-grammaire, Ph.D. thesis, CERIL, Université Paris 7, 1993.

[16]Ristov S, Lauc D. A system for compacting phonebook database, 25th International Conference on Information Technology Interfaces ITI03, Cavtat, Croatia, June 2003. Kalpić D,Hljuz Dobrić V (eds.). 155 – 159.

[17] Ristov S. A Note on Indexing DNA and Protein Sequences, 6th International multiconference Information Society IS03, Vol A: Intelligent and Computer Systems. Ljubljana, Slovenia, October 2003. 121-126

[18] Andersson A, Nilsson S. Improved Behaviour of Tries by Adaptive Branching Information Processing Letters 1993; 46(6): 295-300