Efficient in-memory data structures for n-grams indexing

Daniel Robenek, Jan Platoš, and Václav Snášel

Department of Computer Science, FEI, VSB – Technical University of Ostrava
17. listopadu 15, 708 33, Ostrava-Poruba, Czech Republic
{daniel.robenek.st, jan.platos, vaclav.snasel}@vsb.cz

Abstract. Indexing n-gram phrases from text has many practical applications. Plagiarism detection, comparison of DNA of sequence or spam detection. In this paper we describe several data structures like hash table or B+ tree that could store n-grams for searching. We perform tests that shows their advantages and disadvantages. One of neglected data structure for this purpose, ternary search tree, is deeply described and two performance improvements are proposed.

Keywords: n-gram, ternary tree, B+ tree, hash table

1 Introduction

N-gram is a sequence of elements, i.g. words in document or words in phrase. These n-grams are used within text operations or text comparisons. It mainly goes about finding plagiarisms, spam detection or comparison of sequences of DNA.

The first problem that occurs within a text comparison is the extraction of n-grams itself. It is generally solved by floating window from the beginning to the end of the document. By extraction it is needed to eliminate duplicated n-grams, store the frequency of their appearance or their position in the document for further comparison.

After finishing the extraction it is needed to look up in the n-grams database. Searching has to be quick even though the amount of data is within gigabytes. Sophisticated data structures were invented for this purpose to provide effective access to searching.

In the following article, the use of the ternary search tree (TST) is described as a data structure to search n-grams. There are tests made to compare ternary search tree to the other commonly used data structures for indexing n-grams.

V. Snášel, K. Richta, J. Pokorný (Eds.): Dateso 2013, pp. 48–58, ISBN 978-80-248-2968-5.

2 Related Work

The text n-grams extraction is the first part needed for the future use. We are not interested about all n-grams but the specific ones that occur in text at least m-times [8]. It's because we're comparing similarity of documents, respectively the mostly repeated parts of them. In case of huge texts such as 1.5T TREC ClueWeb-B, the use of the ordinary data structures is, such as hash table or search trees, mainly ineffective because the amount of the data cannot be stored in the RAM. Hard drive can be used as a temporary storage where the preprocessed data can be stored [4]. The second option is to utilize structures like a B+ tree or Hash table to manage this amount of data [6].

Within the extraction is also mainly stored the information about n-gram position in the document. To save space, it is appropriate to store this information without redundancy. The use of double indexing for this case was shown within data collections PROTEIN-10M, PROTEIN-100M and PROTEIN-1G. Due to the size of the index was reduced 1.9 to 2.7 times and the search speed increased up to 13 times [5].

One opportunity how to process the n-grams is to store complete text of this n-gram in a data structure [3]. Effective tool for storing the data is for example the ternary search tree [10] in which every node stores information about one n-gram character. As shown by tests on collections Google WebIT and English Gigaword corpus is the data structure fast enough [3].

However, storing whole n-grams in a data structure considerably increases memory requirements. For this case it is better to use two data structures where the words in n-grams are at first converted to unique numbers and only after that the numbers are processed by data structure [1,6]. The most used data structure to map the words to numbers in n-grams is the hashmap [11]. The hashmap is, thanks to its properties, fast enough and memory effective to convert words to numbers. It is ideal in cases where there is beforehand known the word count.

To store n-grams or the words indexes contained in them is widely used B+ tree [1]. It is no wonder because this data structure was designed to search effectively also with regard to the lack of the memory. In every cell of the B+ tree is stored whole n-gram, which is used for comparison during the search process [2].

This attitude was tested on data collection WebIT 5-gram corpus, which contains over 88GB data separated to collection of unigrams to 5-grams. Thanks to word indexing and the use of B+ trees it was managed to store the whole data collection on 598 MB of memory [1]. In this case there is no problem to have the data in memory and thus avoid using slow hard drives. The creation of the indexes for 5-grams itself takes approximately an hour but it lasts only 2 seconds to look up 1,000 5-grams.

One of the key requirements to look up n-grams is the opportunity to use wildcard placeholders, for example when is suitable to look only for particular similarity. When indexing both words and n-grams is first necessary to find a range of words in the first

index. However, this is only possible when the indexes are sorted with the words. If this case is fulfilled, it is easy to look up using data structures like B+ tree [1].

3 Data Structures

There is a huge amount of data structures, which use the pair <*key, value*> to store data. They are mainly called the map. The selection of ideal data structure is not quite easy task. Mainly there is also need to account the type of data, which will be stored simultaneously with the data structure concept.

The array of pairs <*key, value*> can be presented as the easiest data structure. To find the required element it is needed to go through the whole array of elements or the half in the average case. This access is, however, waste of the computing resources.

For faster search the binary search can be used. To use it, we need that the array of elements is sorted. In case of adding one element in the array there is need to move the half of the array elements in average.

There were more complex data structures invented, which are far more effective when inserting new element in the array or looking up one from the array. These will be described in the following paragraphs.

3.1 Hash Table

Hash table is a data structure, which associates the value of the key with the required value. The straight access in the array is used to get the value. The hash is used as the index, which is computed from the key.

Algorithm of hash computation can be easily deduced[1]. It has to have many properties, which ensure that this data structure will be enough effective. The key requirement is, that the probability of the same hash appearance is minimal for the given data. Furthermore, the resulting hash has to be in a range of the array size. It is computed by the modular arithmetic [7].

The data table is composed by array, whose elements are the pairs <*key, value*>. However, there is a pointer to the pair stored more often. It is appropriate because the hash table is not always filled, so in these cases the free cells would only occupy memory.

In case that the given hash exists in the data table but for the different key, there are two methods to solve such a collision. The first of these uses a concatenation of stored pairs to form a linked list. In case of looking up there is every hash of the key tested until the agreement occurs or the end of the table is reached.

[1]For the following tests there is the djb2 algorithm used to hash the text

The second attitude is so called open addressing, which computes an alternative position for the given hash up to time when the position is free. In this case there is need to go through all of the alternative space until the given key is found. For this attitude the table has to be larger than the count of the elements.

Whereas the hash function is used for indexing, the n-grams can be indexed by only ordering unigrams one by one. The hash is then computed out of these concatenated unigrams.

3.2 B+ Tree

B+ tree is a tree structure outgoing of B-tree. The only main difference is that B+ tree has values stored only in leaves. Every node of the tree contains the array of the keys and the array of the pointers to the following node.

Using the sequence searching or binary search, the pair of keys is found, which limits the search key. Its index is used to found the next node. This attitude is used to get to the leave, which contains a reference to the value of the given key.

Requirements to B+ tree can be summarized to the following 4 points:

- Root has *N* children at maximum
- Every node besides root has at the maximum *N* and at the minimum *N/2* children
- Data are stored only in leaves
- All the leaves has equal level, they are in the same depth

By fulfilling these requirements the tree structure is formed. This structure is always balanced. The advantage of tree structures derived of B-trees is, that by storing the data to the hard drive, the size of the node can be adapted to the hard drive sector size. N-grams can be stored as sequences in B+ trees. The disadvantage of this attitude is the need always to compare the same prefixes of sequences during the key comparisons, which decelerates the search itself.

3.3 Ternary AVL Tree

Binary search tree (BST) is a data structure, which consists of vertices, which always contains the value and edges. It exists one root element and every vertex contains two edges to the following vertices. One edge points to the vertex, whose value is always bigger and the second edge points to the vertex whose value is smaller.

To look up an element it is enough to start at the root and with the simple comparison go through the tree to the required result. Thanks to this, the searching in the tree achieves an average complexity $O(log_2 n)$, where n substitutes the number of the tree elements.

During the tree creation the undesirable situation can happen due to miserably ordered data when the linked list is created instead of tree. For example when the data are ordered ascending by value, the tree is created in which every node has only right child. The created tree has complexity of searching defined as *O(n/2)* in average. Mainly this extreme case does not occur but unbalanced tree has far worse time for look up of elements than a balanced tree. This problem can be solved using one self-balancing tree, for example AVL tree [9]. It goes about binary search tree, which in addition fulfills the condition that the length of the left and the right subtree of node differentiates by 1 at maximum. This is ensured by subtree rotation when inserting new node when needed.

This access increases the time severity when inserting and deleting nodes but it ensures more effective searching when inserting unordered data. Storing n-grams can be done similarly as in case of B+ trees. It means that every node would contain whole n-gram and the text of these n-grams would be compared when searching. But still remains the problem when, at minimum, the identical prefix of the given keys must be compared in given node. Ternary search tree is an adjusted version of binary search tree where every node of the tree contains except two links to the following nodes also one more link. This link points to the root of the next ternary search tree, which contains only a part of the key without the prefix which defines the superior tree.

For example, if we would like to index in ternary search tree the letters “ab”, the first ternary search tree would contain the letter “a” and also contain the link to the second one with the letter ”b”.

As ternary search tree can also be unbalanced implying worse search times, it is suitable to combine this data structure with the self-balancing idea. For example the self-balancing ternary AVL tree can be built, which would have suitable properties for future use.

In some cases there is a need to store created tree on the disc. There is one simple solution. The tree itself is stored in the one dynamic array, so by storing this array and some necessary variables the backup is done. To quickly create original tree is just necessary to allocate new array, copy stored one there and copy stored variables.

3.4 Hybrid AVL Tree

Using the ternary search tree for storing whole n-grams can involve problems with the depth of some binary search trees. We made a test with collection of 3-grams[2], where the counts of the search trees in ternary search tree were detected. Table 1 shows result distribution of binary trees. It was found that more than 3 % of binary trees has depth greater than 4. In addition these binary trees are one of the most used binary trees in the ternary tree.

[2]Random lines extracted from Web 1T 5-gram, 10 European Languages Version 1 collection

One option how to stop creating the binary search trees in case of occurrence so deep tree is to change this tree to trees with multiple roots. This is attained by small hash table, which is placed instead of root of the binary search tree. As a hash function is used only modulo to obtain sufficient search speed. By test was found that adding this hash table is effective at the moment when the depth of the tree is greater than 4.

Table 1. Depth of binary trees in ternary tree

BST depth	1,000,000 n-grams	5,000,000 n-grams	10,000,000 n-grams
1	4,770,413	30,204,256	62,114,350
2	239,038	1,188,767	2,522,381
3	103,559	489,166	1,010,700
4	42,342	196,679	390,481
5	15,693	73,598	139,014
6	4,478	24,277	4,7099
7	983	7,821	14,392
8	82	847	1480

3.5 Double Ternary Search AVL Tree

If we use n-gram as n-tuple of words, the considerable redundancy occurs. Thanks to the redundancy the consumption of the working memory considerably increases and operations made with these n-grams are also slow.

If the n-gram can be divided to more words, the words and the n-grams composed of these words can be indexed independently [1, 6]. Indexing of the words is meant the conversion from the text form of word to the numeric value. Occurrence of the word in the text is repeated and therefore it is suitable have these numbers unique only when the words vary. Thanks to this, the redundancy can be avoided.

If the words are converted to numbers, the n-gram itself does not consist of the text now, but of the indexes of numbers. With the use of this knowledge, the two previously described ternary search trees can be joined. During inserting the n-grams to the tree it has to be divided by a set of symbols. Resulting words are inserted into the first ternary search tree, which stores the unique values of the word.

Every vertex in this tree stores one character as a key. After getting a complete list of indexes of words, the second tree is filled. In the second tree, every vertex stores the index of the given word as a key.

The search process is similar. If no word is found in the first tree, the given n-gram surely not exists. If every word exists, the search process continues to the second tree. Similarly as by ternary search tree, also by double ternary search tree the self-balancing AVL and hybrid AVL trees can be used.

4 The Average Time and Space Complexity of Data Structures

Before the testing itself it is suitable to describe the time complexity and space complexity of described data structures. In the following article the M would represent the n-gram count and N would represent the number of the words in n-gram and P would represent the average length of the word in an n-gram.

Hash table using a good hash function and enough big array has the time complexity for the insert operation of $O(N*P)$. This is true when a hash function goes through the whole sequence during the hash computing. If the element count is unknown, there can occur the situation, when the allocated array of hash table is insufficient and it decreases the efficiency of the data structure. At the moment there is need to reallocate an array of hash table and recalculate the hash for all the elements.

A hash table includes keys and a table with the pointers to these keys. This table is usually greater than the elements count, so the size will be twice as large as the elements count. In the case of sequence storing, there have to be next to the key, the pointer to the possible value. The conclusion is that the space complexity is defined as $O(2*N + M*N*P + M)$.

Searching in B+ tree can be divided in two parts. In the first part there is need to find the right link in the node. Whereas the values are ordered, the binary search can be used to search the value. In the second part we move to the next level of the tree. By searching the tree the time complexity is defined as $O(\log_B(M) * \log_2(B))$ where B defines the number of keys in the node.

In case of ternary search tree where every node contains one character, link to the left and to the right subtree and the link to the subtree which represents the next character of the sequence. The time and also the space complexity can be hardly exactly determined because it mainly depends on the count of the identical prefixes.

5 Data Structures Comparison

In this section, tests of previously mentioned data structures will be performed. All data structures will be tested on n-gram collection, which contains 1,000,000 of n-grams[3]. Moreover there are four collections, 2-grams, 3-grams, 4-grams and 5-grams. This allows us to discover behavior of data structures to different size of n-grams. Average length of the n-gram of each collection is shown in Table 2. Only in Double hybrid AVL tree and Double ternary AVL tree there is used technique of separate word and n-gram indexing, that was previously mentioned.

[3]Data collection can be found at http://www.ngrams.info/free.asp

Table 2. Average length of n-grams

	2-gram	3-gram	4-gram	5-gram
Avg. length [characters]	14.01	16.77	20.40	24.65

There will be compared seven implementations of data structures. Each test was performed several times for better accuracy. Tests were performed on computer with 2.0Ghz Core 2 Duo processor and 4GB RAM. Measurements were performed by per-process timer from the CPU.

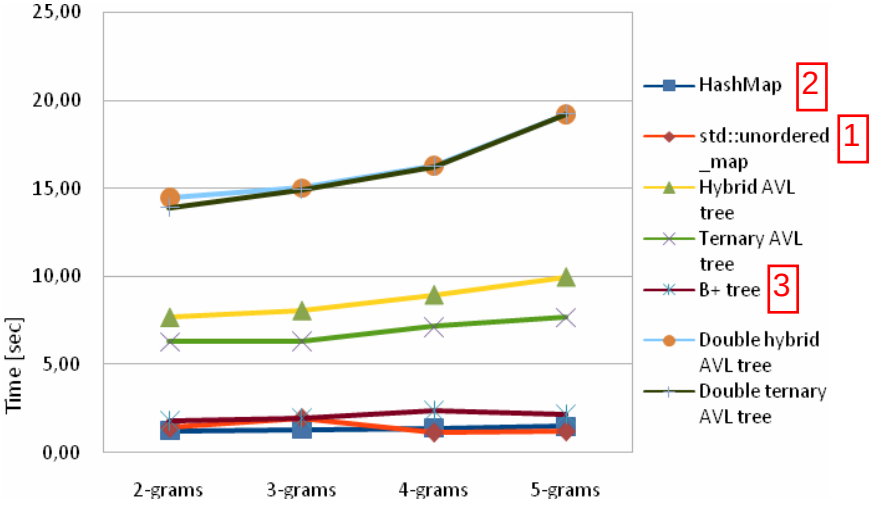

Fig. 1. Insert time comparison

5.1 Comparison by Time of Inserting

This test measures time that is necessary for n-gram insertion. Each data structure is separately created and filled up.

The result on Fig. 1 shows huge difference between ternary tree data structures and the others. This difference may be caused by balancing process due data insertion. This deficiency could be solved by using another type of self-balanced tree, for example red-black tree.

Duration of hash table reallocation seems to be negligible. The worst impact of n-gram size is visible on double trees.

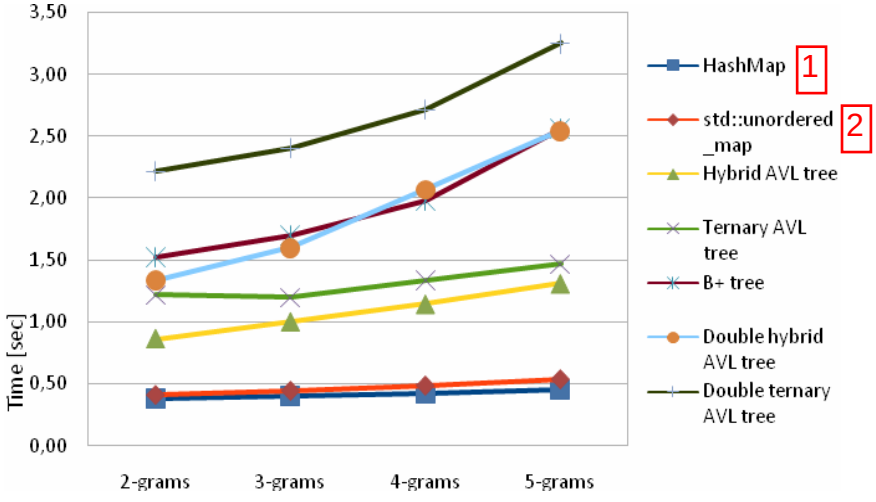

Fig. 2. Comparison of search time

5.2 Comparison by Time of Searching

Search is performed after n-gram insertion. All n-grams from collection are found and time is measured. The result is shown in Fig. 2.

Results shows the best performance of hash table data structures. But on hash table can't be efficiently performed search with wildcard placeholder.

Hybrid variants of ternary trees shows great speed-up. Hybrid AVL tree has up to 29% better search performance than Ternary AVL tree. And Double hybrid AVL tree has up to 40% better search speed than Double ternary AVL tree. Double hybrid AVL tree has comparable results to B+ tree.

The result of B+ tree[4] shows significant increase of search time depending on n-gram size. Moreover, the search time for 5-grams is about 0,41s greater than its insert time. This can be partially caused by necessity of complete look up through the tree in case of search. In the other hand, size of tree increase during insertion.

[4]The implementation of used B+ tree can be found on http://panthema.net/2007/stx-btree/

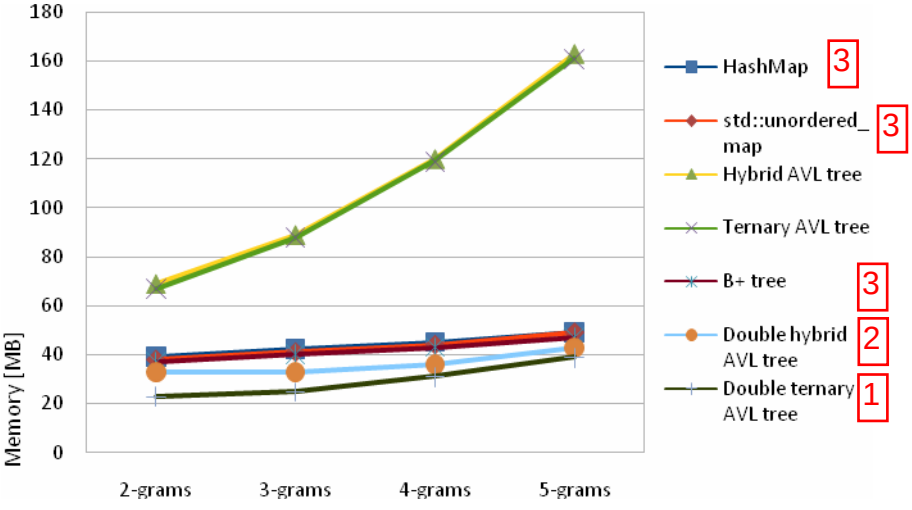

Fig. 3. Comparison of memory consumption

5.3 Comparison by Space Complexity

Last performed test is focused on memory consumption of data structures. Result shows difference of allocated memory before and after data insertion.

Fig. 3 shows large memory requirements of Ternary AVL tree and Hybrid AVL tree, mainly at 5-grams. This is caused by percentage shorter identical prefix of n-grams. This problem solves double variant of this trees.

Double ternary AVL tree has even lesser memory consumption than hash table and B+ tree. The memory consumption is about 40% smaller.

6 Conclusion

This paper described data structures for n-gram indexing such as Hash table, B+tree and ternary trees. Moreover, several approaches for improving ternary search tree efficiency was proposed. The using of the hash table at the 4% nodes of ternary tree with large depth improved the tree efficiency by 40%.

Moreover, this paper shown that the separate indexing of words and n-grams greatly reduced the space complexity. The space complexity of 5-grams reached only 25% originally required memory of ternary tree. The following work will be pointed to the detailed research of data structures for indexing words and n-grams separately. There will also be tested data structures with the requirement to search with the wildcard placeholder. Related to this will be explored data structures for indexing multidimensional data.

Acknowledgement: This work is supported by Grant of SGS No. SP2013/70, VŠB - Technical University of Ostrava, Czech Republic.

7 References

1. Hakan Ceylan and Rada Mihalcea. 2011. An efficient indexer for large N-gram corpora. In Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: Systems Demonstrations (HLT '11). Association for Computational Linguistics, Stroudsburg, PA, USA, 103-108.
2. Douglas Comer. 1979. Ubiquitous B-Tree. ACM Comput. Surv. 11, 2 (June 1979), 121-137.
3. Michael Flor. US Patent, EDUCATIONAL TESTING SERVICE, Princeton, NJ (US). Systems and Methods for Optimizing Very Large N-Gram Collections for Speed and Memory [patent]. United States. Patent Application Publication, US 2011/0320498 A1. Dec. 29, 2011.
4. Samuel Huston, Alistair Moffat, and W. Bruce Croft. 2011. Efficient indexing of repeated n-grams. In Proceedings of the fourth ACM international conference on Web search and data mining(WSDM '11). ACM, New York, NY, USA, 127-136.
5. Min-Soo Kim, Kyu-Young Whang, Jae-Gil Lee, and Min-Jae Lee. 2005. n-gram/2L: a space and time efficient two-level n-gram inverted index structure. In Proceedings of the 31st international conference on Very large data bases (VLDB '05). VLDB Endowment 325-336.
6. Kratky, M.; Baca, R.; Bednar, D.; Walder, J.; Dvorsky, J.; Chovanec, P., "Index-based n-gram extraction from large document collections," Digital Information Management (ICDIM), 2011 Sixth International Conference on , vol., no., pp.73,78, 26-28 Sept. 2011
7. B. J. McKenzie, R. Harries, and T. Bell. 1990. Selecting a hashing algorithm. Softw. Pract. Exper.20, 2 (February 1990), 209-224.
8. J. Pomikálek and P. Rychlý, “Detecting Co-Derivative Documents in Large Text Collections,” in Proceedings of the Sixth International Language Resources and Evaluation (LREC’08). Marrakech, Morocco. European Language Resources Association (ELRA), 2008, pp. 132–135.
9. Robert Sedgewick, Algorithms, Addison-Wesley, 1983, ISBN 0-201-06672-6, page 199, chapter 15: Balanced Trees.
10. David E. Siegel. 1998. All searches are divided into three parts: string searches using ternary trees. In Proceedings of the APL98 conference on Array processing language (APL '98). ACM, New York, NY, USA, 57-68.
11. Justin Zobel, Steffen Heinz, and Hugh E. Williams. 2001. In-memory hash tables for accumulating text vocabularies. Inf. Process. Lett. 80, 6 (December 2001), 271-277.