A String Prefix Dependent Dictionary Structure Based on Hashing and Indexing

Abbas Mohsen Al-Bakry
College of Information Technology, Kufa University, Al-Najaf, Iraq
Abbasm.albakry@uokufa.edu.iq
Marwa Kadhim Al-Rikaby
College of Information Technology, Babylon University, Babylon, Iraq
marwaalrikaby@gmail.com

Abstract

Dictionaries are fundamental components, typically, of every Natural Language Processing system. We proposed a hashing-indexing method to speed up looking up process inside dictionaries. It is a reconstruction of English dictionary of about 300,000 lexical entries using a combination of hash function and an indexing table. This hash function achieves random access according to tokens prefixes, index table keeps track of where the packet referred by the hash key is starting and how long it is. The hash function is perfect but not uniform; indexing is based on two levels, both are stated depending on similarity among tokens.

Key words—hash function, indexing, dictionary, looking up.

الخلاصة

إنّ القواميس مكوّن رئيسي في معظم انظمة معالجة اللغة الطبيعية وهذه الطريقة تعتمد على الفهرسة ودالة هاش لتسريع عملية البحث في ألقواميس فهي عبارة عن عملية اعادة هيكلة لقاموس يحتوي على حوالي 300,000 كلمة بواسطة دمج دالة هاش مع جدول فهرسة حيث توفر دالة الهاش وصولا عشوائيا اعتمادا على الحروف الاولى من الكلمة، في حين تحتفظ الفهارس بعنوان بداية الحزمة التي تحتوي على الكلمة المشار اليها بدالة الهاش مع طول تلك الحزمة، كما ان جدول الفهرسة مؤسس على مستويين كلاهما معتمد على التشابه بين الكلمات. ان الطريقة المقترحة مثالية ولكنها ليست منتظمة.

الكلمات المفتاحية :دالة هاش، فهرسة، قاموس، بحث في القواميس.

I. Introduction

"A dictionary is a collection of words in one or more specific languages, often listed alphabetically with usage information, definitions, etymologies, phonetics, pronunciations, and other information" [Niel08], and "what we call an NLP (Natural Language Processing) dictionary is a linguistic data set that lists words and provides information about them in such a way that exploitation in NLP applications is possible"[Éric13].

Hence, a dictionary is a data structure, in its simplest format consists of at least a table with two columns, the first contains the underlying language tokens and the second is a description associated with that token.

It is a collection of distinct searchable strings (or string sequences) extracted from the text. Dictionary strings are usually indexed for faster access. A typical dictionary index allows for exact search and, occasionally, for prefix search. [Leon11]

Indexing of strings fragments has been commonly used already in the 70s and 80s for the purpose of document retrieval [Schu73][Schek78][Will79] and approximate dictionary searching [Ange83].

Key-indexing is an in-memory lookup technique based strictly on direct addressing into an array with no comparisons between keys made. Its area of applicability is limited to integer keys falling in a limited range defined by available memory resources. Hashing helps direct addressing work on keys of any type and range by bringing serial search and collision resolution policies into the equation.[Drof02]

Hash tables are a common data structure. They consist of an array (the hash table) and a mapping (the hash function). The hash function maps keys into hash values. [Knuth73]

An ideal hash function is "one_to_one" function but this feature is inapplicable in cases of dictionaries because every token would require a unique hash key resulting in a huge dictionary size. However, it is not a hard task to collect similar tokens in one packet and referring to its head by a unique hash key using one_to_many hash function.

Distributing tokens within packets uniformly in order to balance search time inside different packets. This feature also can't be implemented in many data bases and NLP dictionary is one of them where it is difficult, if not impossible, to have equivalent packets because of tokens variances.

Indexing mechanisms, on the other hand, are used to speed up data access. Index tables, usually, are smaller than the original tables. They are references to data packets rather than data itself.

The availability of duplicating indexing levels reduced search space to a reasonable amount. But the way in which the data is stored within packets stayed a challenge facing looking up process since it affects the search technique.

Comer and Shen had presented a hash-binary search method which hashes a key into a packet and applies a binary search within the packet. Their method required a unique hash key for every token in the dictionary, worked on a sample English dictionary of 16,949 entries, and assumed that the hashing functions distribute the keys uniformly. [Com79]

This method is proposed as a solution to reduce looking up time by combining random and sequential access techniques. Random access is implemented by using hash function; sequential access comes as a complement by applying it depending on indexing table.

Our method is efficient and fast for static dictionaries because deletion and insertion operations consume time, but this is not a problem in most of NLP systems because dictionaries fundamental operations in such systems, typically, limited in looking up rather than updating contents.

II. Motivation

Natural languages contain a huge number of tokens stored in special purposes dictionaries. According to the variance of applications which work with NLP techniques, dictionaries are varying in content and format. Spell checkers and text correction systems, NL dependent applications, deal with NLP dictionaries, their resistivity to achieve good results is based on the under hand language dictionary content. The content of NLP dictionaries is usually multilingual and of huge size, and its structure is not linear, and ordered entry by entry but has the form of a complex graph.

The dictionary of modern retrieval systems and spell checkers can be very large, partly because they treat some sequences of adjacent space separated words as a single string.[Leon11]

This variance gave dictionaries a specific feature, since containing more tokens makes the under hand system more resistible against spell errors but also huge size became a problem because looking up time became longer.

As a result, developers are doing their best to keep dictionaries as inclusive as possible and directed their works to speed up looking up process by utilizing hashing, indexing, sorting, and search techniques.

We proposed this method to overcome the problem of looking up strings in a reasonable time through exploiting hash functions to achieve random access which in turn reduces the consumed time.

III. The Proposed Method

We exploited hash functions features and indexing mechanisms to reduce looking up time to a reasonable amount; since "textual data have a limited range of values (the alphabet size) and unlimited dimensionality (the string length); construction of the hash function differs for textual and geometric data" [Skala10], our hash function was merged with indexing to utilize both textual and numeric characteristics.

Search algorithms that use hashing consist of two separate parts. The first step is to compute a *hash function* that transforms the search key into an array index. Ideally, different keys would map to different indices. This ideal is generally beyond our reach, so we have to face the possibility that two or more different keys may hash to the same array index. Thus, the second part of a hashing search is a *collision-resolution* process that deals with this situation.[Sedg14]

Our method consists of two fundamentals: the first is a structure consists of hashing procedure and indexing mechanism; and the second is a looking up procedure.

A. Proposed dictionary structure

A reconstruction was applied on the standard dictionary in order to reduce the required time for finding a target token in it. The goal of this structure is to find a way that is capable of directly find the packet index in the tokens table. By "directly" we mean: only one operation is needed to get packet head address.

- The alphabet of the underlying language (English in our system) is specified according to the set of characters used in tokens:
  Uppercase Letters A-Z
  Lowercase Letters a-z
  Numbers 0-9
  Some special characters ' / - _ .
  This number of symbols can be smaller if we assign 0 instead of any other number 1-9 because in the case of looking up a number (in our system) there is no need to know the value of that number, the purpose of looking up process is to catch the associated tag with that number. Also, numbers are infinite; if we want to associate a tag with each number then we need to account all numbers and store them in the dictionary resulting in an infinite dictionary.
- The resulted alphabet is:
  ∑={ A, B, …, Z, a, b, … , z, 0, ', / , _, - , .}
  This makes the maximum number of symbols in the alphabet less than 64 and hence we can represent every symbol using only 6 bits.
  From this point our hash function emanated. It is a mapping from alphabetic encoding into a numeric form by converting the first three characters from the token prefix to their new encoding according to Table.1, and then applies Eq.1:

$$Index = E(C_1) * 64^2 + E(C_2) * 64 + E(C_3) \quad \ldots (1)$$

  Where E( ) is a function to convert one character to its new encoding which is a numeric value within interval (0,63) taken from Table.1.

- The minimum value of Index is 0 when token prefix is "AAA" and 262143 when the three symbols are set to the last symbol in the alphabet "***". In our system this entry is not used since the last 6 entries are free (||∑||=58).

- As a result the dictionary structure is composed of two tables: *Reference Table* and *Tokens Table*.
  Reference table consists of 2^18 entry, an entry for each different 3 symbols combination. As shown in Fig.1.

Table.1: Alphabet Encoding

Symbol	Code	Symbol	Code	Symbol	Code
A	0	B	1	C	2
D	3	E	4	F	5
G	6	H	7	I	8
J	9	K	10	L	11
M	12	N	13	O	14
P	15	Q	16	R	17
S	18	T	19	U	20
V	21	W	22	X	23
Y	24	Z	25	A	26
b	27	C	28	D	29
e	30	F	31	G	32
h	33	I	34	J	35
k	36	L	37	M	38
n	39	O	40	P	41
q	42	R	43	S	44
t	45	U	46	V	47
w	48	X	29	Y	50
z	51	'	52	/	53
-	54	_	55	.	56
0	57	whitespace	58	*	59
*	60	*	61	*	62
*	63				

- The resulted alphabet is:
  ∑={ A, B, …, Z, a, b, … , z, 0, ', / , _, - , .}
  This makes the maximum number of symbols in the alphabet less than 64 and hence we can represent every symbol using only 6 bits.
  From this point our hash function emanated. It is a mapping from alphabetic encoding into a numeric form by converting the first three characters from the token prefix to their new encoding according to Table.1, and then applies Eq.1:

$$Index = E(C_1) * 64^2 + E(C_2) * 64 + E(C_3) \quad \ldots (1)$$

  Where E( ) is a function to convert one character to its new encoding which is a numeric value within interval (0,63) taken from Table.1.
- The minimum value of Index is 0 when token prefix is "AAA" and 262143 when the three symbols are set to the last symbol in the alphabet "***". In our system this entry is not used since the last 6 entries are free (||∑||=58).
- As a result the dictionary structure is composed of two tables: *Reference Table* and *Tokens Table*.

Reference table consists of 2^18 entry, an entry for each different 3 symbols combination. As shown in Fig.1.
Every record in this table holds the following info:

- Record index which is equivalent to a hash value of a token prefix.
- ***Base*** field holds the address of the packet B which contains all the tokens that are equivalent in their 3-symbols prefix (which its hash value led to this record).
- ***Limit*** field holds the size of B.

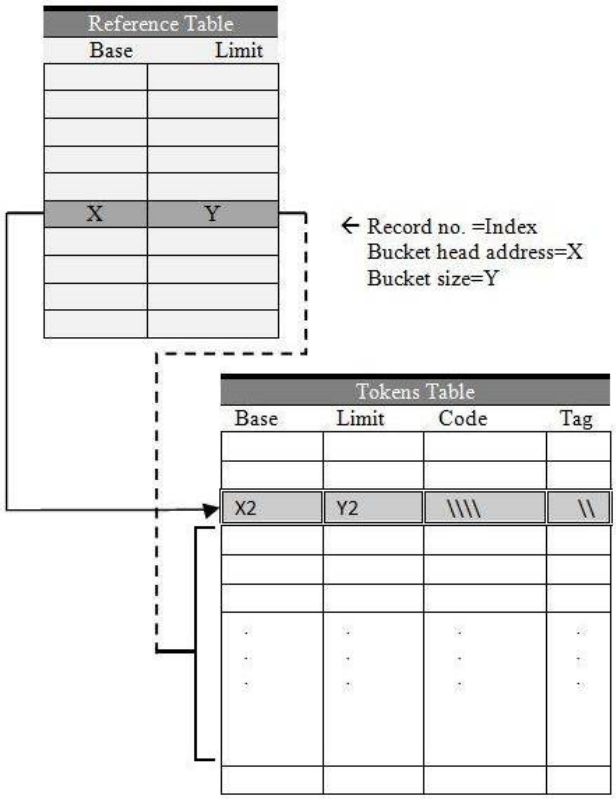


Fig.1. Dictionary Structure

Algorithm1: Token Hashing
Input: *English token (finite string over ∑)*
Output: *packet head address in which the input token may rely.*
Step1: *set variables C_1,C_2, and C_3 to the input token prefix.*
Step2: *apply Eq.1 on C_1,C_2,and C_3.*
Step3: *go to reference table at the record indexed with* ***Index****.*
Step4: *examine* ***Base*** *field, if it is set to a value greater than -1 return this value as the packet head address, else return fail.*
End.

Tokens Table, where the real dictionary tokens are stored, consists of four fields: Base and Limit fields for search purpose, while Code and Tag fields are the standard components of the dictionary (Code contains tokens and Tag contains the associated POS tag with those tokens), but there is a hidden structure constructed as primary and secondary packets.

Tokens are divided into two levels packets depending on prefixes similarity; on the first level packets, tokens are identical with their first three characters and each token may be a head of a secondary packet.

Primary packets contains tokens those are identical in their 3-symbols prefix, which can be hashed and accessed randomly through Reference Table, and each token may be a head of a secondary packet where all tokens within it are identical in their 6-symbols prefix.

Every primary packet stores information about its tokens according to the main table fields:

- Code field holds a token.
- Tag field holds the tag of the token stored in Code field.
- Base field holds an address.
- Limit field is the most important one among the four, since its value determine whether the address in Base is useful or not and the token in Code is a head of a secondary packet.

  If Limit value is zero, then the token in Code has no associated packet, i.e. this token is unique with its 6-symbol prefix and the address stored in Base is useless.

  If Limit value is greater than 0 then it represents a secondary packet size where all tokens that are equivalent to the current token in their 6-symbols prefixes are stored.

  Otherwise, the current token belongs to a secondary packet and no more expansions are found.

B.Looking up Procedure

Once the primary packet head address becomes in hand, looking up process can be started within that packet.

According to fig.1, If the target token is not the one stored at record X then search will be continued using information in the Base and Limit fields; while Base holds the secondary packet head address, Limit holds the size of that packet.

Tokens within secondary packets are identical in their first six characters; this representation reduced search space to a reasonable amount, and made the worst case search time shorter, where target token is not found in the dictionary.

Algorithm2: Looking up a Tag of target token

Input: *Target Token, Primary Packet Head address, Primary Packet Size.*

Output: *tag of input target token.*

Step1: *Set primary packet information*

X=head address.

Y=packet size.

Step2**:** *Examine X:*

1) *if X<0, then the token is not found in the dictionary*

2) if X>=0, goto record at index X and lookup every token in the packet until finding a full match with the 6-characters prefix of the target token.

2.1) if the current token = target token return its Tag.

2.2) if the current token <> target token, set X2=Base field (secondary packet head address) and Y2=Limit field (secondary packet size).

3) if no match is found, then target token is not in the dictionary.

Step3: *Examine Y2:*

1) if Y2 <=0 then no similar tokens, the secondary packet is empty and the address stored in X2 is useless.

2) if Y2>0, goto X2 and start looking up the target token completely not only its prefix:

2.1) if there is a full match then return the target token Tag.

2.2) if no full match is found, the target token is missed.

End.

IV. Case study

As a case study we would take a sample token to look it in the reconstructed dictionary. The structure would be declared in more details through mapping the token prefix using the hash function and looking for it in both primary and secondary packets.

Let us take the token "Adrian" as an example:

A. Calculate hash value

Index= E(A)*64^2+E(d)*64+E(r)

= 0 * 64^2 + 29*64 + 43

= 1899

"***Index***" is the record number (in *Reference Table*) where Base field holds the address of the packet head in which the token "Adrian" may be stored.

Base=625=primary packet head address
Limit=12= size of the primary packet

Hence, the search would be directed to that packet.

Next, go to *Tokens Table.*

B. In *Tokens Table,* depending on primary packet head address, move to record indexed by 625 and check if the current token (Adrian) is equivalent to the head token in the 6-characters prefix (handled in Code field).

On the right of fig.2 , whole primary packet consisting of 12 record as referred by:

{ Reference Table: Index Record : Limit Field }

It starts from index 625 ending at 637, all tokens start with the same 3-symbols prefix as "Adrian".

We met our target after 9 comparisons, notice that the Limit field value is 13 which is the size of a secondary packet holds tokens start with the 6-symbols prefix "Adrian" and its head address is the value of Base field (639), this secondary packet is shown in fig.3.

Index	Reference	
	Base	Limit
.	:	:
1895	579	9
1897	589	5
1896	600	3
1897	604	14
1898	-1	-1
1899	-1	-1
1900	625	12
1901	-1	-1
.	-1	-1
	657	3
	:	:

Address	Tokens			
	Base	Limit	Tag	Code
625	638	0	N	Adrammelech
626	638	1	N	Adrastea
.	639	0	N	Adrea
.	639	0	N	Adrell
.	639	0	N	Adrenalin
.	639	0	N	Adrestus
.	639	0	N	Adria
633	639	0	N	Adrial
.	639	13	N	Adrian
637	652	1	AN	Adriatic
	653	1	N	Adriel
	654	3	N	Adrien

Fig.2. Case Study, Target Token ="Adrian"

Address	Tokens			
	Base	Limit	Tag	Code
639	-1	-1	S	Adrian I
640	-1	-1	S	Adrian II
.	-1	-1	S	Adrian IV
.	-1	-1	S	Adrian V
.	-1	-1	S	Adrian VI
.	-1	-1	N	Adriana
.	-1	-1	N	Adriane
.	-1	-1	N	Adrianna
.	-1	-1	N	Adrianne
.	-1	-1	N	Adriano
.	-1	-1	N	Adrianople
651	-1	-1	S	Adrianople red
	-1	-1	N	Adrianopolis

Fig.3. Secondary Bucket

V. Results

The proposed method reduced looking up time is via three stages:

1. After calculating Hash value, if the reference record at H( ) is set to -1 this means the target token is not in the dictionary; reducing total search time to only one operation.
2. If step 1 was passed, the search would be limited within primary packet only; if the target prefix (of length 6) is not matched with any token in that packet then the target token is not found; reducing search within the size of the primary packet.
3. If step 2 was also passed, there is only one further packet to be looked up; secondary packet, where all tokens are equivalent to the target in 6-characters prefixes, is the last search space and the failure in this step consuming:

1+ the size of the primary packet + the size of secondary packet

i.e.

$cost = O(Y1+Y2)$

Where

Y1 is the primary packet size

Y2 is the secondary packet size

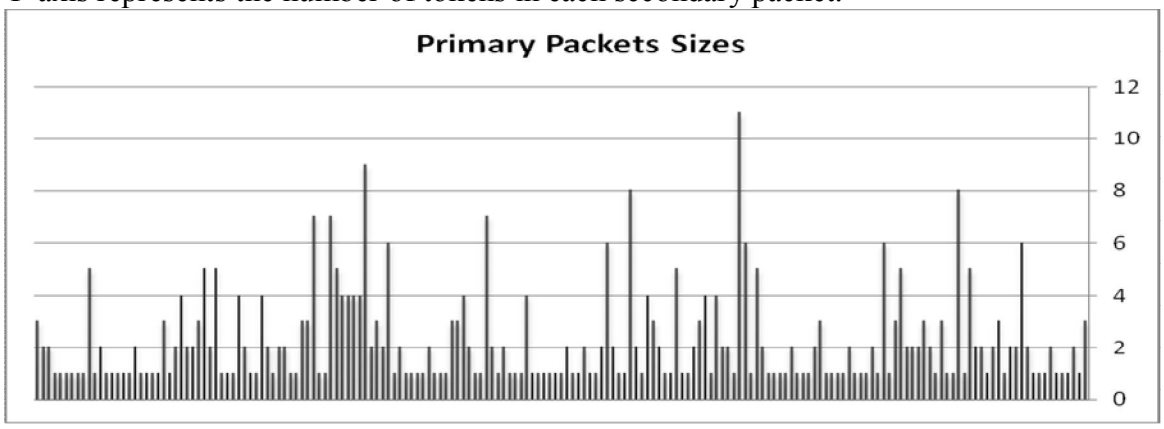
as a worst case.

While the best case is:

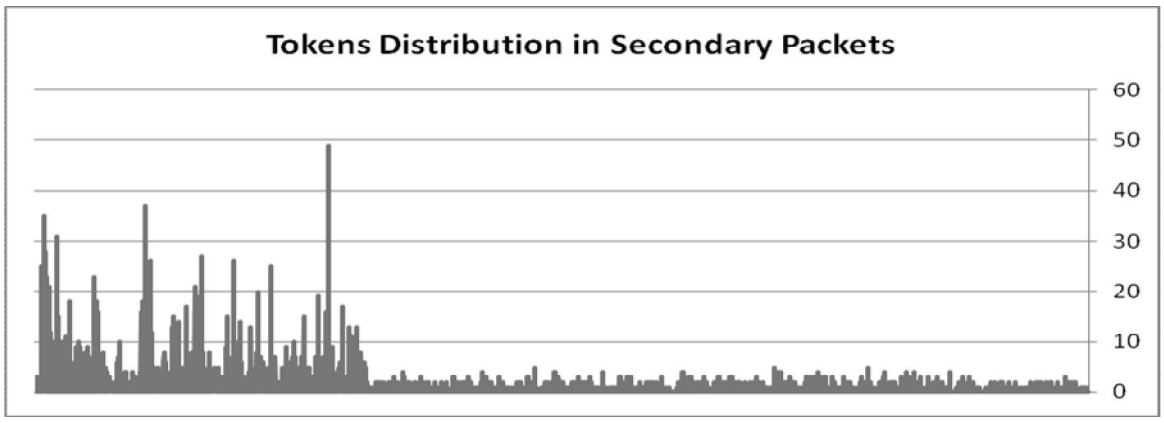
1. For successful search, one operation is required for hashing and another for matching with the head of the primary packet.
2. For invalid search, also one operation is required when hash value leading to an empty packet.

In both situations, search cost is ***O(1)***.

Figure.4 shows the sizes of primary packets, where the Y-axis represents the length of the packets which in turn represents the number of secondary packets heads in each primary packet. Figure.5 shows how tokens are distributed in secondary packets, the Y-axis represents the number of tokens in each secondary packet.



(fig4: Variance of Primary Packets Sizes)



(fig5: Tokens Distribution within Secondary Packets)

VI. Conclusion

The proposed method reconstructs the basic dictionary into a more efficient structure where less time is needed to lookup a token within it. The method exploits hash function features to build a directly accessed reference table as a starting point in the looking up process toward the second part of the dictionary database in which all tokens are stored in the format of packets. These packets are constructed in a two level structure where similar tokens stored together in a secondary packet addressed by a head token, the last is a part from a primary packet and this primary packet can be accessed directly from the reference table.

Hashing achieved random access; packets are examined using sequential search because tokens are stored abstractly without any secondary keys. Even it may seem to be slow, but efficient. It reduces the reconstructed dictionary size and packets loading time since all packets can be found in memory at the same time.

We hope for enhancing the method via using faster search technique. Depending on tokens themselves, far away from associating a hash key with each token, binary or any other faster technique can decrease looking up time.

References

Angell R.C., Freund G. E., and Willett P., 1983, "*Automatic spelling correction using a trigram similarity measure*", Information Processing and Management 19, 4, 443–453.

Comer, D. and Shen, V.Y., 2008, *"Hash-Binary Search: A Fast Technique for Searching an English Spelling Dictionary"*. W. Lafayette: Purdue University, Computer Sciences Department.

Dorfman, P.M., 2002, *"Table Look-Up by Direct Addressing: Key-Indexing -- Bitmapping – Hashing"*, Jacksonville: CitiCorp AT&T Universal Card.

Éric L., 2013, *"Dictionaries For Language Processing Readability And Organization Of Information",* Universidade Federal do Espírito Santo, Université Paris-Est.

Knuth D., 1973, *"The Art of Computer Programming"*, Volume 3: Sorting and Searching, Chapter 6.4. Addison Wesley.

Leonid B., 2011, "*Indexing Methods for Approximate Dictionary Searching: Comparative Analysis*", ACM Journal of Experimental Algorithmics, New York.

Nielsen and Sandro, 2008, "*The Effect of Lexicographical Information Costs on Dictionary Making and Use*". Tübingeb: Gunter Narr, *Lexikos* 18: 170–189.

R. Sedgewick and K. Wayne, May 05 2014,"hash tables", http://algs4.cs.princeton.edu/34hash/.

Schek H. J., 1978, "*The reference string indexing method"*, in Proceedings of the 2nd Conference of the European Cooperation in Informatics, Springer-Verlag, 432–459.

Schuegraf E. J., 1973, "*Selection of equifrequent work fragments for information retrieval*", Information Storage and Retrieval 9, 12, 697–711.

Skala, V., 2010, Hrádek, J., and Kuchař, M., *"New Hash Function Construction for Textual and Geometric Data Retrieval"*. Corfu, Greece: University of West Bohemia,Department of Computer Science and Engineering.

Willet P., 1979, "*Document retrieval experiments using indexing vocabularies of varying size*", II. Hashing, truncation, digram and trigram encoding of index terms, Journal of Documentation 35, 4, 296–305.