Comparison and Analysis of Fast Searching Techniques In English Dictionary

Sarthak Jain - 191IT145
Information Technology
National Institute of Technology Karnataka
Surathkal, India 575025
Email: sarthak94511@gmail.com

Yash Gupta - 191IT158
Information Technology
National Institute of Technology Karnataka
Surathkal, India 575025
Email: guptayash1104@gmail.com

Rishit - 191IT141
Information Technology
National Institute of Technology Karnataka
Surathkal, India 575025
Email: rishit.191it141@nitk.edu.in

***Abstract*—Searching in a large amount of data is a time consuming process and is one of the most frequent task performed by a server. Reducing the time taken to search is of huge profit both for service providers and consumers. In this paper we analyze and compare methods to make data searching faster. In Trie Based storage of data, greedy algorithm can be applied to make searching significantly faster. Apart from faster searching, trie can also be used for suggesting autocomplete words based on the prefix entered. If the data is stored in a traditional database, to improve searching speed caching of the searched words is done using several caching algorithms.**

I. Introduction

Data is the most crucial asset of this age. Massive databases consisting of many terabytes of data are used by many private firms, government agencies, banks, and more companies. Searching and querying these databases is one of the most frequent tasks servers perform, and it consumes a large amount of computation time. Thus a fast searching approach becomes essential if the minimum response time of the server is desired. In this paper, we analyze and compare different searching techniques on dictionary database and finally select the most efficient searching approach to query the words and provide their meaning and usage in response.

In today's world, for most daily tasks such as transactions and other activities requiring user authentication, fast searching of user account information and data related to that account becomes necessary. Life is very fast-paced, and delay of any sort is not desirable by anyone. With the increasing size of data, a trivial searching algorithm, i.e., searching the whole data one by one, is very time-consuming and inefficient. To improve the average query time, storing some entries(caching) using a fixed size cache with different eviction/replacement algorithms is implemented. Following eviction policies are considered - FIFO(First In First Out), LRU(Least Recently Used), LFU(Least Frequently Used), RR(Random Replacement). All these eviction policies are highly efficient greedy algorithms but the concept of caching is dynamic programming paradigm. Memorizing the previous results so that they need not to be calculated again. Suppose the data is stored in lexicographical order. In that case, Binary search can be one efficient algorithm for fast searching, taking logarithmic time to find a word in data, but we can still make some improvements to improve the search time further.

Trie or prefix tree is a data structure that stores the set of strings. The distinguishing characteristic of the trie is, using the searching algorithm on data stored in a trie, the data retrieval can be done in linear time complexity. If the length of the searched word is k, then the time taken to search is of O(k) time complexity. It is the most optimized approach to search for a term in a large number of words. Along with fast searching, trie has another use essential for any search requirement where a user types the search query. That feature is autocompleting / suggestions based on the prefix of the word that the user has entered. This feature helps in many scenarios—one being when the user doesn't know the exact spelling of the word. This auto-complete feature can also be implemented using regular expression queries in standard database implementation. Still, it is not very efficient as it scans every entry in the database to find matching prefixes, which can be time-consuming in case of a large amount of data. Whereas the trie based approach is much optimized and faster performing because trie is nothing but a prefix tree, so the tree is scanned in the direction of the prefix, and then all the children of that node are explored to get all the possible words in data that can be formed from that prefix.

To provide a real-life use to the research and study done during this project, a dictionary web app where users can enter a word to search and as they are typing, suggestions based on the prefix entered will be shown, and the user can select a

word from them and then search. An HTTP request will be sent to the server, and in response, the meaning of the word, its usage, and a photograph related to that word is shown. So the web app provides a fast searching of meaning and usage of the word along with the photograph related to that word. To provide the image, Unsplash API is used, which provides the link of image based on the searched word.

The report is organised in the following structure: Introduction, Literature Survey, Problem Statement, Methodology, Result and Analysis, Conclusion, Acknowledgement, Individual Contribution, References

II. Literature Survey

For successful implementation of our project we referred following resources:Through this paper[1] by Jiang S,Ding X,Chen F, we leant how important is reduction of latency to improve the type of experience that user gets.Higher latency can result in loss of interest of user from application.This paper also sites how we can effectively exploit localities and can successfully create buffer structure to hold the data.Effective Management Schemes are also described in this paper.

Through this paper by [2] by Marshall B., Welborn C.R. we learnt how databases are effectively constructed ,the algorithms and study of data structures associated with databases too are described in well organised manner in this paper.The steps to perform analysis on the databases are also cited in this paper.This paper laid the path for us to effectively analyse the backend part of our project.

Through this paper by [3] by Abrams M, Standridge CR, Abdulla G, Williams S, Fox EA we learnt what are the potential rise in performance that we can gain by implementing caching based algorithms efficiently.This paper also helped us in learning about limitations of particular algorithms This led us to analyse the performance of our algorithms by varying the size of cache buffer for all algorithms on the requests of word made by the users.

Through this paper by [4] by Andersen P, Petersen N be learnt about methods of performing efficiency analysis on these types of algorithms.We also learnt how variation of size to a extent improves the performance of the algorithms but increasing the size beyond a certain point will increase the time complexity of operations and thus making the efficiency of implementation of these algorithms is lower and this also impacts the user experience and increases the latency because of increment in complexities.

Through this paper by [5] by Kastaniotis G, Maragos E, Dimitsas V, Douligeris C, Despotis DK we learnt about object storage in data structures efficiently and also we leant about analysis on object replacements.The method of analysis are mostly statistical and graph based.We learnt how time can be effectively measured for monitoring performance of the algorithms.

Through this paper by [6] by Chang C, McGregor T, Holmes G We learnt about Least Recently Used algorithm implementation.This paper shows steps involved in the implementation of the algorithm though this paper does not show any comparison with other algorithms and also does not perform any analysis based on varying the size of cache buffer. The time complexity for implementation of Least Recently Used algorithm is also calculated and analysis is performed in other aspects too.

Through this paper by [7] by Selvakumar S, Sahoo S-K, Venkatasubramani We learnt about Least Frequently Used algorithm implementation.This paper shows steps involved in the implementation of the algorithm though like the previous paper,this paper also does not show any comparison with other algorithms and also does not perform any analysis based on varying the size of cache buffer.The time complexity for implementation of Least Recently Used algorithm is also calculated.

Through this paper by [8] by Smirlis YG, Maragos EK, Despotis DK. various method for deployment of application is shown and also the methods for performing efficient analysis ,graphical analysis based on recorded time and also how load can be reduced by choosing the best algorithm for replacement of data.We also learnt about how analysis can be performed mathematically for these types of problems, which are real time in nature and where application may undergo heavy load.

Through this paper by [9] by Psounis K, Prabhakar B. We learnt about another algorithm called random replacement algorithm where an object in cache is randomly picked and is replaced with the current data.Though this paper does not show comparision of random replacement algorithm with other algorithms neither graphically nor mathmatically. We also learnt about the steps of performing time complexity analysis for random replacement algorithm.

Through this paper by [10] by Siriopoulos C, Tziogkidis P we learnt about general statistical methods to monitor performance of algorithms efficiently.We also learnt about how to effectively represent data in form of graphs for these types of algorithms.mathematical analysis general steps are also mentioned in this paper which helped us to calculate complexities of the algorithms used in this project.

III. Problem Statement

To compare and analyse different fast searching techniques including Caching algorithms and Trie based implementation on an English dictionary

A. Objectives

- Implementing LRU, LFU, FIFO, and RR caching algorithms for improving average search time in a data query.
- Implement Trie based data storage and algorithms for fast searching and auto-completion based on prefix
- Comparing and analyzing the run-times of searching in these algorithms and plotting the results
- Use the best algorithm and create a web server which can fastly provide the meaning of word
- Create an easy to use web app where user can enter a word to search and get the meaning and usage of word

IV. Methodology

To speedup the search process, two different approaches have been used.

TABLE I
SUMMARY OF LITERATURE SURVEY

Authors	Methodology	Merits	Limitations	Additional Details Based on your project (eg. Dataset Used)
Jiang S., Ding X., Chen F., Tan E., Zhang X., DULO [1]	scheme to exploit both temporal and spatial locality	Importance of exploitation in simple ways possible	Only therotical aspects are covered	Real life use cases
Marshall, B., Welborn, C.R. [2]	complex algorithm on advanced data structures	methods of analysis of complex algorithms mathematically	Limited to certain types of algorithm	Algorithmic analysis
Selvakumar S, Sahoo S-K, Venkatasubramani V [7]	least frequently used algorithm implementation	analysis of least frequently used algorithm	Comparision to others not presen	Algorithm Implementation
Psounis K, Prabhakar B. [9]	implemenmtation of random replacement efficiently	analysisof complex algorithm mathematically	Comparison with others is not made	algorithmic analysis

1) For data stored in a traditional database, speedup can be gained by using caching.
2) Trie based storage of data and then using greedy trie-retrieval algorithm.

A. Trivial Searching

In case when data is stored in a traditional database, for any search query, we need to scan the complete database in order to get our desired entry. If the amount of data stored is large, the time taken for searching the whole database is large. It leads to decrease in performance of system and may lead to loss of profit for both company and clients. If the data is stored in lexicographical order, binary search, a divide and conquer algorithm can be used to search an entry faster. Database indexing is another way to decrease the search time and that is performed internally by DBMS using B/B+ trees, but in this paper we are exploring other options to improve search time.

B. Speedup Using Cache

The technique of keeping copies of files in a cache, or temporary storage area, so that they can be retrieved more rapidly, is known as caching. A cache, in technical terms, is any temporary storage site. Cache has low storage space but very high performance speed. When the requested data can be located in a cache, it's called a cache hit; when it can't, it's called a cache miss. Cache hits are served by reading data from the cache, which is faster than repeating a result or reading from a slower data storage; hence, the faster the system runs, the more requests that can be served from the cache. Instead of retrieving data from a backup store, a cache might store data that is computed on demand. Memorization is an optimization technique that saves the results of resource-intensive function calls in a lookup table, allowing later operations to utilise the saved results and avoid recalculation. It is related to the **dynamic programming** algorithm design methodology, which can also be thought of as a means of caching.

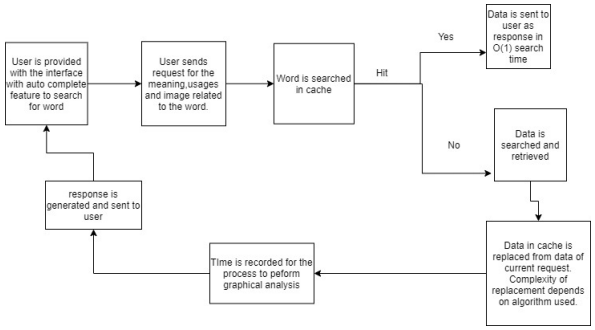

Fig. 1. Searching methodology using caching

Algorithm 1 Searching methodology using Cache

Input: search word
Output: meaning of word

```
Initialisation: result

if word is present in cache then
    return cache[word]
else
    result = Search word in database
    Store the (word,result) pair in cache
end if

return result
```

As stated in Algorithm [1], in a cached server, whenever a request is made, it is checked if the result of the request has previously been calculated and memorized in the cache. If the result is already calculated, then the response id directly sent after retrieving it from cache in constant time complexity. In the case if the entry is not found in cache, then the trivial search is performed on the database and the results are stored in the cache before returning it to response. This is the fundamental concept of dynamic programming. Memorizing the results for future use.

Hashmap is used to serve as a cache as it has O(1) insertions and searching. Caches are high performing but their storage space is less, so they get full and then there is no space for more insertions. To deal with this issue, each cache has some eviction/replacement policy. Whenever the cache

is full, this eviction algorithm is used to delete a specific entry from the cache. Several replacement algorithms are there which can be used to evict an entry and make space for new entry. In this paper, four such algorithms are discussed namely FIFO(First In First Out), LRU(Least Recently Used), LFU(Least Frequently Used), RR(Random Replacement). The first 3 (FIFO, LRU, LFU) comes under greedy algorithm paradigm.

1) FIFO: This is the simplest cache eviction algorithm. It follows the techniques used in a queue Fist In First Out, i.e. when the cache is full, the oldest entry in the cache is deleted to make space for new one.

Algorithm 2 FIFO Cache Replacement

Input: search word, result
Ensure: store the (word,result) pair in cache
Initialisation: cache = hashmap(cache size)
1: **if** cache is full **then**
2: Delete the first entry in cache
3: store cache[word] = result
4: **else**
5: Store cache[word] = result
6: **end if**

Algorithm [2] states the steps taken to evict an entry in case the cache is full. If the cache is not full, the new entry can be directly inserted in the cache hashmap, whereas when the cache is full, the oldest entry in the hashmap is tracked and deleted. After successful deletion, the new entry is inserted in the cache. It is a simple algorithm but is useful in certain scenarios when the probability of search of all the words is nearly same. If the hashmap doesn't support the tracking of entries in order, then a queue data structure can be used along with the hashmap. The queue can track the order of insertions of words in cache and then the first entry of the queue can be delete everytime a space is required for new item.

2) LRU: LRU stands for Least Recently Used. As the name suggests, in this cache replacement algorithm, the item which is least recently used is deleted/evicted from the cache to make space for new items. In FIFO, we only bother about cache updation when there is a cache miss but in the case of LRU cache, we update the cache even in the case of cache hit.

If cache hit, then the item searched is moved to a position(head of linked list) where it indicates that it is the most recently used item. So whichever word is searched, that entry moves to the head of the linked list. Thus the last position(tail of the linked list) denotes the Least Recently Used item and it is deleted whenever the cache is full and space for new entry is desired.

Algorithm 3 LRU Cache

Input: search word, result
Ensure: store the (word,result) pair in cache
Initialisation:
1: Linked List - to store meaning and usages of word, hashmap(word,address) - to store the address of Node where details of word is stored
2:
3: **if** word in hashmap **then**
4: Get address of Linked List node using hashmap
5: Move word's linked list node to the head of the linked list
6: **else**
7: **if** cache is full **then**
8: Find the Least Recently Used(present at the tail of linked lis)
9: Delete that word from Linked List and hashmap
10: **end if**
11: Insert a new Node containing meaning of word at head of linked list
12: Store the new node's adderss in the hashmap
13: **end if**

As stated in Algorithm[3], when a search query is given to server, if the word queried is already present in the cache, then we have to make this word as most recently queried. To do that, it's Linked List Node is moved to the head of the linked list. In this way we can get the Least recently used item at the tail of the linked list.

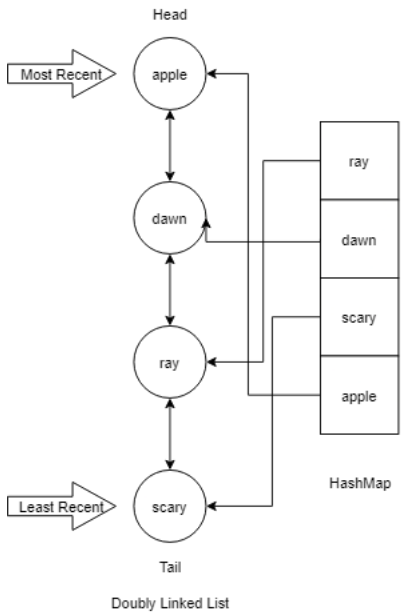

Fig. 2. How LRU Cache tracks the least recently used item.

In case of cache miss, it is searched using the naive approach and after the results are obtained it is inserted in the cache. In the case when cache is full, the least recently used item(present at the tail of linked list) is deleted and its entry

from the hashmap is also deleted. LRU is a greedy algorithm as at each eviction it greedily chooses the Least recently used item and delete it. The algorithm performs well and provides good amount of cache hits in scenarios where the probability of a word already searched recently is more than other words.

3) LFU: LFU stands for least frequently used. For most of the purpose, this is the optimal choice for caching algorithm as it evicts the least frequently used item whenever the cache is full. In cases like a dictionary, it works good and provide high cache hits as there are some common words which people will search and their frequency will keep increasing and they will always be in the cache thus faster search. Similar to LRU, in LFU cache is updated not only in cache miss but also in cache hit. In case of a cache hit, the frequency of searched word is increased(internally, it is moved to the linked list which has 1 frequency more than its current frequency).

To implement LFU cache eviction algorithm, heaps can be used but the operations then have a logarithmic time complexity. But this can be reduced to constant time operations using Hashmaps and linked lists. Two hashmaps are used, 1 containing the (word,pointer) pair which provides the address of the node of the corresponding word. Second hashmap stores the (frequency,pointer pair). For each frequency, different linked list is maintained. Whenever a deletion is required, item is deleted from the tale of the linked list which has least frequency.

Algorithm 4 LFU Cache

Input: search word, result
Ensure: store the (word,result) pair in cache
Initialisation: Linked List - to store meaning and usages of word, hashmap(word, address) - to store the address of Node where details of word is stored, hashmap(frequency, address) - to store the head of linked list with given search frequency

if word in hashmap **then**
    Get address of Linked List node using hashmap
    Move word's linked list node to the head of the linked list of the next frequency
else
    if cache is full **then**
        Find the Least Frequently Used (present at the tail of the lowest frequency of linked list)
        Delete that word from Linked List and hashmap
    end if
    Insert a new Node containing meaning of word at head of linked list with frequency 1
    Store the new node's address in the hashmap
end if

The algorithm [4] is LFU + LRU as after finding the list lowest frequency elements, the element is deleted from the tail of that Linked List. Similarly when an item moves to higher frequency linked list, it is inserted at the head position of the linked list. This combination in one of the most optimal cache replacement algorithm.

4) RR: Random Replacement is another cache replacement algorithm when the requests we are dealing with is purely random. In this algorithm, whenever a cache miss occurs and cache is full, to insert the new entry, randomly and element is chosen and deleted from the cache and new element is then inserted in the cache. No updation is done in case of cache hits. This algorithm is implemented solely using a hashmap. Whenever needed to delete, a random integer is generated in the range [0,size) and then after traversing to that item, it is deleted from hashmap. This algorithm takes O(n) time for eviction while all other algorithms take O(1) time.

Algorithm 5 Random Replacement Cache

Input: search word, result
Ensure: store the (word,result) pair in cache
Initialisation:
hashmap(word,address) - to store the (word,meaning) pair as cache

if word is not in cache **then**
    if cache is full **then**
        Select a random entry in cache and delete it
        Store the word meaning pair in cache
    else
        Store the word meaning pair in cache
    end if
end if

C. Trie Based Implementation

1) Building Trie: Trie is is used to store the english words of the Oxford dictionary for faster search and lookup.The trie nodes, have 26 children nodes each node holding the next charcater of the English Alphabet and it contains a boolean value defining whether the node considered is a leaf node or not.

The root node of the trie is stired with the vakue NULL to denote that the trie is initially empty.The word to be inserted is then stored charcater by character in a tree like structure.Firts the index of the current character is found by mapping the current charcater to be considered to the numbers 0 through 25.The boolean value of this node is set accordingly- if the character is the last charcater of the word then the boolean value is set to true else it is set to false.Once the leaf is reached the word is successfully inserted.The same algorithm is then repeated for all the words of the english dictionary dataset thus resulting in the formation of the trie.

Algorithm 6 Building trie of english words

Input: Dataset containing the english words in the oxford dictionary

Output: Trie built using the english words of the oxford dictionary

```
    Initialisation
    Node
    (
       Node *child[26]
       bool End
    )
    Node *p = new Node;
    Node.End = false;
 1:
 2: for i = 0 to 26 do
 3:    p.child[i] = NULL
 4: end for
 5:
 6: Node *p = root;
 7:
 8: for i = 0 to word.length do
 9:    index = key[i] - 'a';
10:    if (!p.child[index]) then
11:       p.child[index] = getNode
12:    end if
13:    p = p.child[index];
14: end for
15:
16: p.End = true
17:
18: return Trie
```

Algorithm 7 Searching a word in a trie

Input: Trie containing the english words

Output: Word searched and the corresponding meaning

```
    Initialisation
 1: Node *p = root;
 2:
 3: for i = 0 to word.length do
 4:    index = key[i] - 'a';
 5:    if (!p.child[index]) then
 6:       return NOT FOUND
 7:    end if
 8:    p = p.child[index];
 9: end for
10:
11: Meaning of the word is saerched
12:
13: return wordmeaning
```

2) Searching in Trie: The word to be searched is soterd in a string .The word is then scanned character by character and we move along the trie in the same fashion visiting each chracter node once.This is done until either the word exhausts and we reach the leaf node or the word 's character doesn't exist at the required position which means that the word is not in the dictionary.The serach time is therefore reduced to O(length of the word) when compared to naive approach which takes O(sixe of dictionary * sixe of words).If the word is found then the corresponding meaning is retrieved fom the dicionary and the meaning is then displayed to the user else the user is warned that the word searched doesn't exisit in the oxford dictionary dataset.

3) Autocomplete: When the user begins searching or the word in the dictionary then he is prompted certain suggestions all of which are such that the currently typed word is prefix of the word suggested.In order to implement this using tries thw word currentky tyoed is first searched using the technique of searching the words in the trie as discussed in the previous paragraph.If the word serach returns a boolean value of false then the no suggestions are sent to the user else the current node is stored in a temporary node.From this node all it's 26 children are visited and is any child is empty the next child node is visited and the enitre trie below this child node is then visited until the a leaf node is visited.On encountering a leaf node the word is stored in an array and then sent to the UI whre the UI displays this array as a drop-down menu giving suggestions to the user.On selecting a specific suggestion the search method is called and the meaning is displayed to the user.

Algorithm 8 Auto Complete using Tries

Input: Trie containing the english words

Output: Suggestion Words for the word searched

```
    Initialisation
 1: Node *root = NULL
 2: for i = 0 to word.length do
 3:    if (!p.child[index]) then
 4:       root = currnode
 5:       break
 6:    end if
 7:    p = p.child[index];
 8: end for
 9: words = []
10: Node *p = root;
11:
12: for i = 0 to 26 do
13:    index = key[i] - 'a';
14:    if (!p.child[index]) then
15:       words.push(currword)
16:       return to original word
17:    end if
18:    p = p.child[index];
19: end for
20:
21:
22: return words[]
```

V. Experimental Results and Analysis

A. Discussion and Complexity analysis

The trie based implementation provided the best performance because of its very fast searching and autocomplete feature. While the time taken for traditional database server to search a word came out to be 65-100 ms, Trie took only 1-4 ms to provide the result of a search query. Same time was taken by a cached server during a cache hit, but cache hit percentages gor all the algorithms was less than 40%, so the average search time was decreased but the time taken during cache miss was still large.

Building Trie

The time complexity of building a trie is O(m*n) where m is the average size of each word and n represents the total number of english words in the trie as Each character of each word is tobe inserted into each node therefore each word needs to be traversed character by character and since n words are to be insertted therefore the time complexity will be O(m*n).

The space complexity of building a trie is O(m*n*k) where m represents the averageword size and n represents the number of words in the dictionary and k is the alphabet size which is 26 here as the english language contains 26 alphabets.Since each node stores a single charcater and there are m*n characters m*n space is required but since each node also contains 26 children nodes the space complexity becomes O(m*n*k).

Searching Trie

The tries are the best data structres for searching as they can reduce the time coomplexity of the searching by large amounts.Here the searching time in trie is O(m) where m represents the average size of the words that are to be inserted in the trie, as each charcatre is compared with the original word and since there are m characters in the word the average time complexity is O(m). The space complexity of searching is O(1) as no space is used once the trie is built for searching.

Naive Search

Time taken for searching a traditional database is O(N*m), where N is the number of entries in the database and m is the average word length. Fig.3 shows the time taken for searching for 150 queries. Time taken ranges from 85ms-250ms.

Using Cache

Fig.4-7 shows the time taken to search in cached servers. Average search time is reduced in cached servers as in case of cache hit, result is available in 1-3 ms which is very fast. For all the algorithms, cache hit percentage was less than 40%. Random Replacement gave the highest cache hits 59/150 while FIFO had lowest 48/150. Random Replacement performed the best because the queries were also made randomly, but for the real life use, LFU will perform better in case of dictionary application as there are some commonly searched words which will always be in cache.

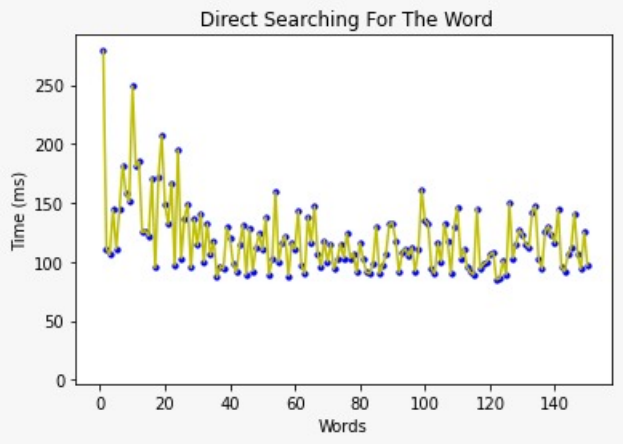

Fig. 3. Time taken for word search without using cache, Avg Time = 119.11ms

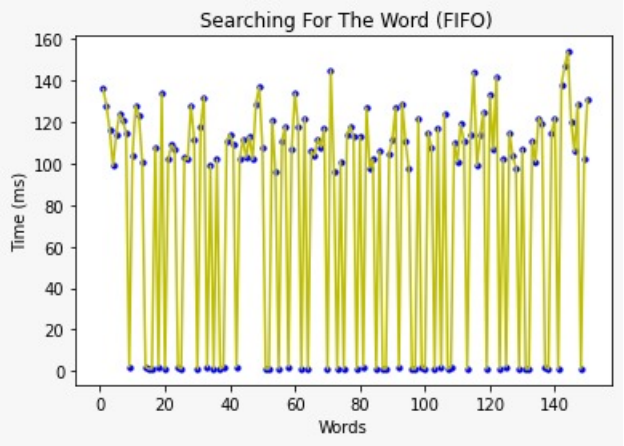

Fig. 4. Time taken for word search using FIFO Cache(48/150 cache hit), Avg Time = 78.7ms

Fig.8 shows the comparison of cache hit percentages of the algorithms used when cache size taken was 20. RR and LFU are the best performing cache eviction algorithms.

Fig.9 shows the variation of cache hit percentage with increasing cache size. As expected, with increasing cache size, cache hits also increase.

Fig.10 and Fig.11 shows the snapshots of the dictionary web app. The backend server uses the trie based implementation for fast search of meaning of word. Unsplash API is used to get a picture based on the searched word.

VI. Conclusion

In the applications where server experiences lots of load like hundreds and thousands of the users in real time it becomes important to reduce latency as much as possible. The reduction in latency not only results in better user experience but also reduces the chances of failure or crashing of application.In order to reduce the work load of the servers efficient caching algorithms can used so that search time on servers can be reduced to greater extent.Efficient caching prevents searching up in database everytime a query is received from an user.

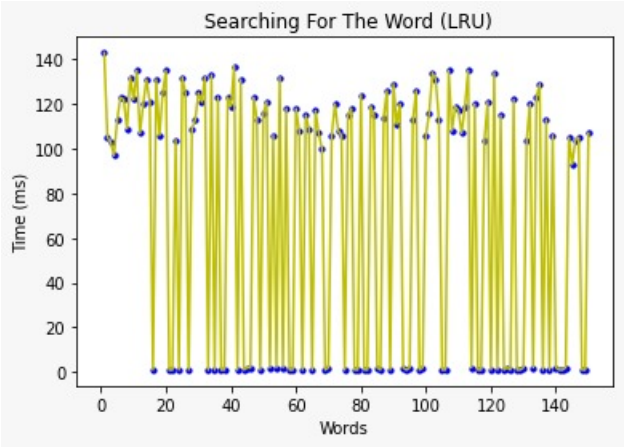

Fig. 5. Time taken for word search using LRU Cache(56/150 cache hit), Avg Time = 74.25ms

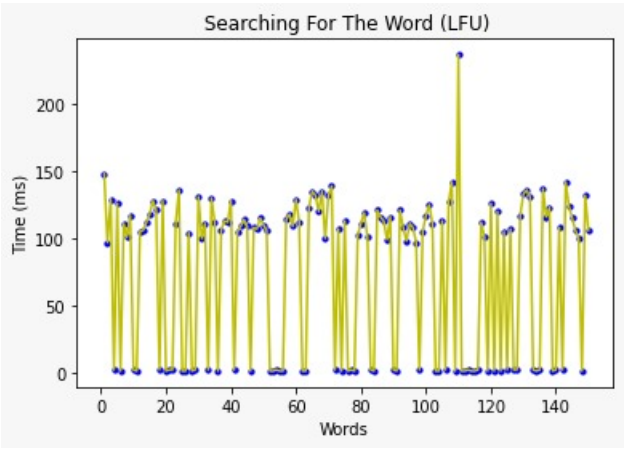

Fig. 6. Time taken for word search using LFU Cache(58/150 cache hit), Avg Time = 72.79ms

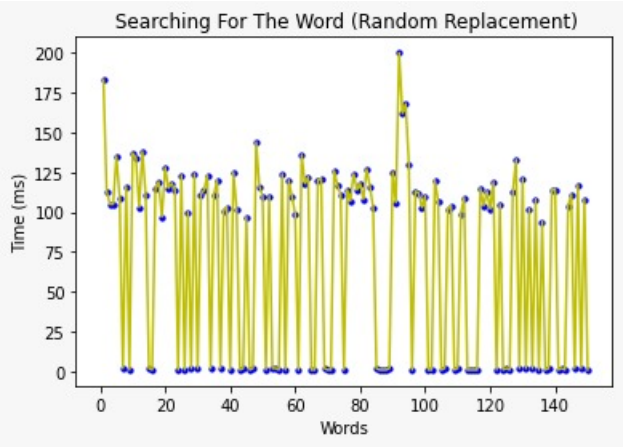

Fig. 7. Time taken for word search using RR Cache(59/150 cache hit), Avg Time = 71.55ms

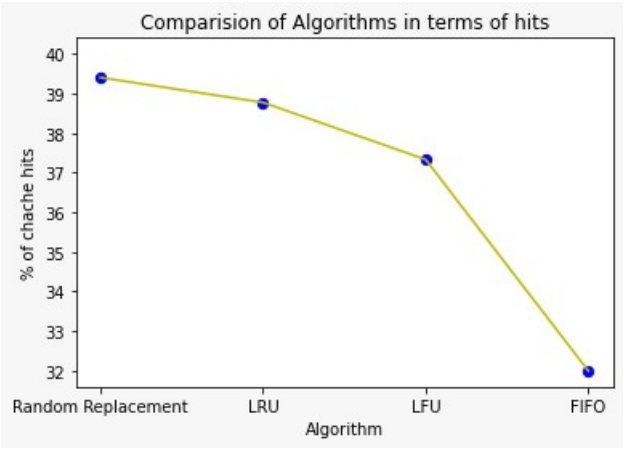

Fig. 8. Cache Hits of algorithms used. Cache size = 20

Dictionay and image search related to given word along with there usages is very important tool which can really help people learning new language and to the people reading complex literatures ,the usages provided along with meanings of the word also improves ones fluency in the language.In this project we tried covering multiple algorithms and tried analysing how efficiently the time is reduced when there is data cached properly on the server.We have also tried implementing dictionary using trie based data structure and analysed insertion,search,etc algorithms on the top of this trie data structure we tried multiple caching algorithms to reduce the search time in the dictionary and provide better experience to the users of the dictionary.The words are stored in the cache of predetermined size and as the users make request to server inorder to find word's meanings,usage first the cache is searched and if the word is among the words searched in past by other users on the server the results are sent directly to the client without searching in the trie or the database based on the implementation being used on the server side.This reduces latency due to searching and improves the user experience.The multiple algorithms used here are least recently used , least frequently used , first in first out,random replacements.

Precisely these algorithms are cache replacement algorithms which helps in changing the data available in the cache according the currently searched words.If the word being searched is not present in cache , cache miss takes place and according to algorithms word is kept in cache for example random replacement algorithm picks the word randomly out of the cache and replaces it with the data (meanings,usages) corresponding the word being searched whereas least recently algorithm picks the word which is not searched by users lately and replaces with the data obtained of the current word.Least frequently used algorithm picks the word which is not searched very frequently among the words present in the cache and replaces with the data obtained of the current word and first in first out picks very first word and the word is removed and current word is inserted in the cache.these algorithms are analysed and graphical comparision is made between all of them and also the comparision is made between these algorithms by variying the size of cache.The effective caching also helps in reduction of the database cost.This caching

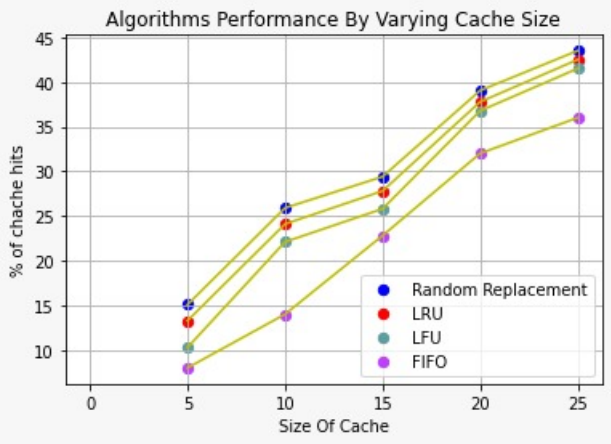

Fig. 9. Percentage of cache hits with increasing size of cache.

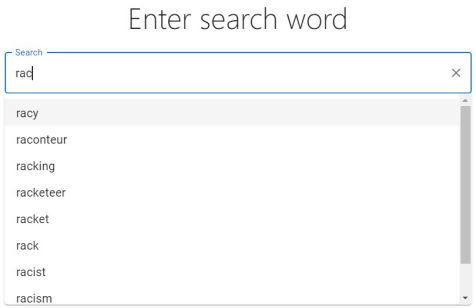

Fig. 10. The auto-complete feature in web app

technique also helps us in the elimination or removal of database hotspots which results in providing more number of resources for a special chunk of data.The load is reduced by a great extenct on backend servers and thus results in increment in throughput of search and read operations performed in database.The main purpose of doing this is to improve user experience and target the reduction in latency inorder to make this learning tool better of currently available options.

The future work in this project involves adding efficient algorithms that can efficiently provide user with the pronunciation of the words and also the translation of word to multiple languages.The public application programming interface can also be developed which can be incorporated in multiple applications since these efficient use of algorithms will make the application programming interface light weight and will improve user experience of that application.Also support for other native languages can be provided thus using this application programming interface words of any real time website can be translated to ones native language and improve his experience.

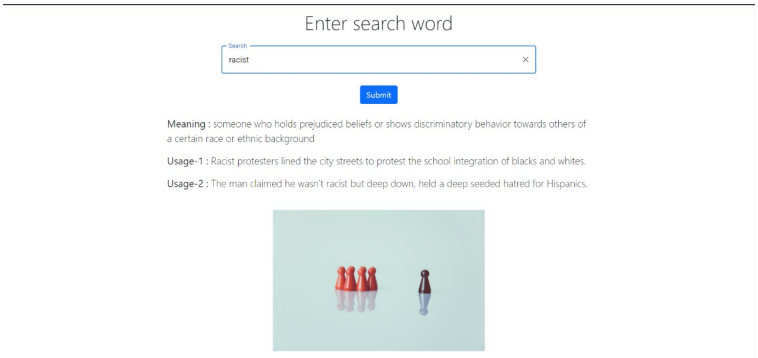

Fig. 11. The Web Interface

ACKNOWLEDGMENT

We would like to express our gratitude to Mr.Ram Mohana Reddy Guddeti for his assistance in making our project successful, along with the IT department, NITK, for providing us the opportunity to work on this project.

INDIVIDUAL CONTRIBUTION

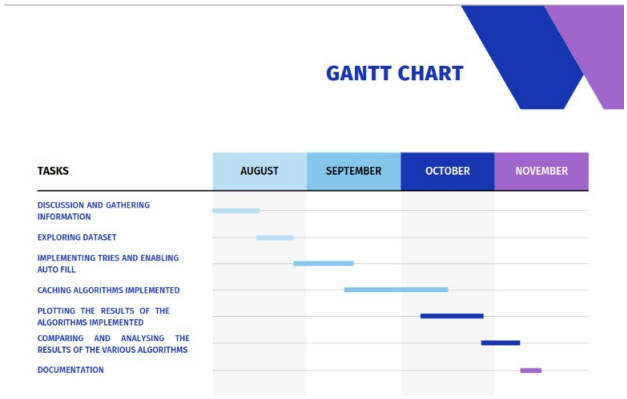

Fig. 12. Gantt Chart

Express js Dictionary data server- Yash ,Rishit
Implementation of Algorithms -Sarthak,Yash,Rishit
Integrating caching in dictionary server-Yash,Sarthak
Building trie based server for search-Rishit,Sarthak,Yash
Trie auto-complete-Yash,Rishit,Sarthak
Analysis of performances of algorithms and plotting graphs-Yash,Rishit,Sarthak
Integrating unsplash api for photo -Sarthak,Rishit
Developing front end user interface-Rishit,Yash,Sarthak

IMPLEMENTED/BASE PAPER

The implementation and analysis of replacement algorithm by Chang C, McGregor T, Holmes G. In: Proceedings of the Asia-Pacific web conference.Hong Kong, China.This paper shows steps involved in the implementation of the single algorithm .Though this paper does not show any comparison

with other algorithms and also does not perform any analysis based on varying the size of cache buffer.

REFERENCES

[1] S. Jiang, X. Ding, F. Chen, E. Tan, and D. Zhang, "an effective buffer cache management scheme to exploit both temporal and spatial locality," *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, vol. 4, 2005.

[2] B. Marshall and C. Welborn, "Understanding a dbms from the inside out," *Journal of Computing Sciences in Colleges*, vol. 27, no. 2, 2011.

[3] M. Abrams, C. Standridge, G. Abdulla, S. Williams, and F. EA., "caching proxies:limitations and potentials," *Proceedings of the 4th WWW conference. Boston, MA*, 2000.

[4] P. Andersen and N. Petersen, "procedure for ranking efficient units in data envelopment analysis," *Management Science 1993*, vol. 18, no. 2, 1995.

[5] G. Kastaniotis, E. Maragos, V. Dimitsas, C. Douligeris, and D. Despotis, "Web proxy caching object replacement: frontier analysis to discover the good-enough algorithms," *Proceedings of the 15th IEEE international symposium on modeling, analysis, and simulation of computer and telecommunication systems. Istanbul, Turkey*, pp. 132–137, 2007.

[6] C. Chang, T. McGregor, and G. Holmes, "The lrun www proxy cache document replacement algorithm," *Proceedings of the Asia-Pacific web conference. Hong Kong, China*, 1999.

[7] S. Selvakumar, S.-K. Sahoo, and V. Venkatasubramani, "Delay sensitive least frequently used algorithm for replacement in web caches," *Elsevier, Computer Communications*, 2004.

[8] Y. Smirlis, E. Maragos, and D. Despotis, "Data envelopment analysis with missing values: an interval dea approach," *Elsevier, Applied Mathematics and Computation*, vol. 1, 2006.

[9] K. Psounis and B. Prabhakar, "Efficient randomized web-cache replacement schemes using samples from past eviction times," *IEEE/ACM Transactions on Networking*, vol. 1, 2002.

[10] C. Siriopoulos and P. Tziogkidis, "How do greek banking institutions react after significant events," *A DEA approach Elsevier, Omega – The International Journal of Management Science*, 2010.