Efficient Fuzzy Search Enabled Hash Map

V. Topac
Department of Computer Science and Information Technology
"Politehnica" University of Timisoara,
Timisoara, ROMANIA
vasile.topac@aut.upt.ro

Abstract- **Hash maps are data structures widely used in modern programming languages like Java for their simplicity and efficiency. When fuzzy string search is needed (like in natural language processing) finding an approximate key match in a regular Java HashMap is a trivial task. It usually requires the brute force method of iterating trough the set of keys and use of string metrics methods. Although this approach works it is time consuming and loses the hashing advantage of the hash map. Another option is to use a different data structure like TreeMap, which is faster, but also have limitations on fuzzy string search. This article presents FuzzyHashMap, an extension to the regular Java HashMap data structure allowing highly efficient fuzzy string key search. Based on object oriented principles this extended hash map uses a custom key that enables different types of pre-hashing functions and different types of dynamic programming algorithms for approximate string matching. Customizable algorithms and settings bring flexibility to this new data structure, making it adaptable to each specific use case. Fuzzy string search performance comparison between FuzzyHashMap and the regular HashMap are presented for both accuracy and time consumption. Results show very good performance for FuzzyHashMap compared to the regular HashMap. Some real use cases for the extended hash map are listed. All the work described in this paper is released as open source, making it easy for the community to use and extend the capabilities of the current implementation.**

Keywords: approximate string matching, fuzzy search, Hash Map, Java

I. Introduction

This article presents a simple to use, yet powerful, fuzzy data structure, implemented on top of a common Java data structure. Modern programming languages like Java have a rich offer of data structures highly optimized and easy to use. For storing and manipulating data in form of key – value mapping the best data structures to use are HashMap, HashTable, TreeMap or LinkedHashMap. TreeMap is usually used when key-ordered collection is needed. LinkedHashMap is mainly used when insertion order is needed. HashMap and HashTable are used when neither key or insertion order matters, but speed and simplicity is important.

When dealing with linguistic text storing, for text collection in the form of dictionaries, the best choice for storing such data is the HashMap [1] data structure. They have the advantage of storing (in memory) a big amount of data and offering almost instant access to the value mapped on the searched key.

The problem with linguistic data, especially natural language processing, is that it is dealing with uncertain information. A method of dealing with this kind of data is using error-tolerant methods like fuzzy string matching. There is a lot of research done in this area. A few examples of usage of this kind of technique are name matching [2], spellchecking and "on-the-fly" type-ahead suggesting systems [3] and most important textual information search in string collections, using a big variety of methods, some described in [4].

In many cases when dealing with text, the back-end data storage solutions are databases. Fuzzy usage in database information manipulation is being worked on for a while, FSQL (FuzzySQL) or SQLf getting closer to standardization [5]. Not the same thing is true when coming to in memory fuzzy data structures. This paper focuses on in memory data structures, usually used in small or medium applications, where speed and fuzziness is needed.

The current version of the data structure presented in this paper, and possible future releases are available as open source at the FuzzyHashMap project web address [6].

II. Methods

A. *HashMap*

HashMap data structures are highly used in modern programming language as Java. They enable storing of key – value pears, offering no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time. The main advantage hash maps offer is high speed when searching elements by key, given by the hash driven search mechanism instead of iteration trough the set of stored data. Furthermore the implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly. Iteration over collection views requires time proportional to the size of the HashMap instance (the number of buckets) plus its size (the number of key-value mappings). Initial capacity and load factor are important parameters for the performance of the hash map operations. However this article is not aiming to advice on improving performance for regular HashMap structures.

There are two main methods for Java objects used as key that have an important impact over the HashMap performance: *hashCode()* and *equals()*. HashCode method is implementing

978-1-4244-7984-9/10/$26.00 ©2010 IEEE

the function that generates the hash code (an identifier number) corresponding to a certain key. The hash code will be used to identify the location where the map entry will be stored. A bad hashing function can lead to bad HashMap performance, if it consumes too much time or if it will generate the same hash code for two different keys. The last case is called a ***collision***. When a collision happens, *equals* method is used to compare the value already stored on a specific location, with the new one. If the keys are different, a new location, linked to the existing entry, will be allocated. The ideal Java HashMap has no collisions.

B. *FuzzyHashMap*

FuzzyHashMap (FHM) adds fuzzy enabled functionality to the default operations available in standard Java HashMap. In Fig. 1 the UML Class Diagram of the FHM implementation is presented. One can see that FHM class extends the Java HashMap class and aggregates FuzzyKey class. FuzzyHashMap class has multiple constructors, to allow customization of algorithms used for hashing or comparing string.

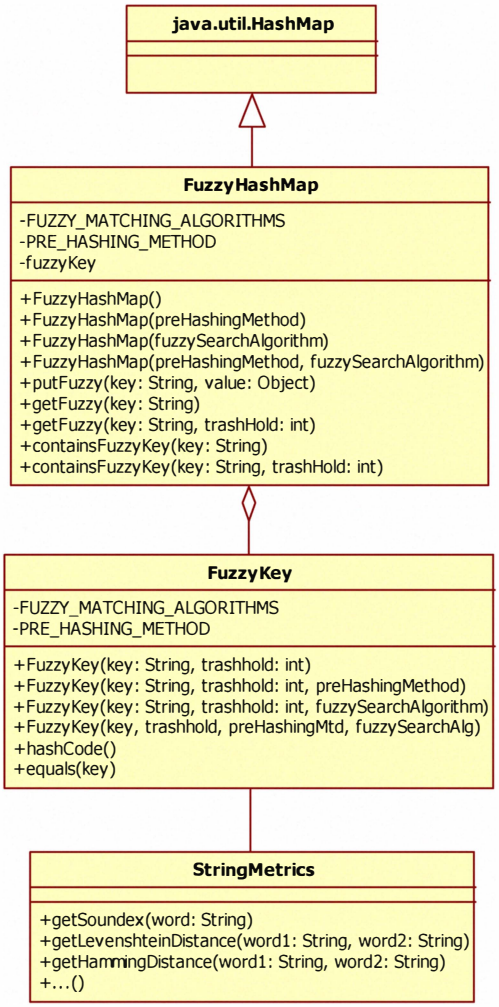

Fig. 1. FuzzyHashMap UML Class Diagram

The most important methods in FHM class are *getFuzzy* and *containsFuzzyKey*, allowing fuzzy query.

The aggregate class FuzzyKey is used to create keys within the FHM class. FuzzyKey class uses algorithms from the associated StringMetrics class to perform string metrics operations.

C. *FuzzyHashMap Functionality*

Further the internal functionality of the FHM is described. Contrarily to regular HashMap, in FuzzyHashMap collisions are being intentionally created and controlled, so that each collision will happen between words from the same "group". The "group" of the words is being dictated by the pre-hashing function.

To achieve an almost native fuzzy search, FHM use fuzzy keys. So for each string key added or searched in the FHM, an instance of FuzzyKey class is created. FuzzyKey class has a string key as member, and most important, it overrides the following default Object methods:

hashCode()
equals().

These methods are called both when adding and when querying the data in the hash map.

The first method called is *hashCode*. The overriding *hashCode()* method performs a pre-hashing function and after that the default Java *hashCode* implementation is used. Here is a pseudo code example of the overriding hashCode function:

```
{
    preHashedKey = performPreHashing(key);
    hashCode = hashCode(preHashedKey);
    return hashCode;
}
```

Several pre-hashing functions are implemented in the current version. The most efficient ones were:

Substring
Soundex

When using *substring*(0, 4) pre-hashing function, a group will contain words that are starting with the same four letters; here is an example of substring function:

substring("Washington",0,4) = "Wash"

Using substring as pre-hashing function will make the FuzzyHashMap have a similar behavior to Java TreeMap, but the inner structure is different and the entries are not being ordered.

The other type of pre-hashing function is *soundex* algorithm [7], which will group words by English phonetic similarity. Soundex algorithm associates an alphanumerical code to a given word, the resulting code having the fist letter equal to the word, and then three numbers calculated according to the letters in the word. An important note is that *soundex* resulting code has always the size four. Here is an example of soundex code:

soundex("Washington") = "W252"

Performing pre-hashing and calculating the hash code is done whenever an item is added or searched in the hash map.

Method *equals()* is called whenever a query operation is done, and only in case of collision for adding operation.

Put operation (adding elements).

In case of adding operation, the *equals* method is used for collision handling. If the hash code value for the new item represents a location already used, then the *equals* method is used to check whether the existing item is equal to the new one. If positive, than the new item will be ignored, else, the new item will be saved at a location linked to the existing item location. In FHM this situating will occur very often, due to the pre-hashing function. A very important note is that in FHM, for adding operations, by default the equals method will not allow fuzziness. This is done to avoid losing similar elements when they are added in the hash map.

Get operation (searching elements)

When searching items, after the hash code was calculated, the item stored at the resulted hash code value is checked for similarity by using *equals* method. Notice that this time, equals method will allow fuzziness, in order to obtain approximate matches. The accepted similarity level can be controlled by a threshold value stored in the FuzzyKey object. If no similar match is found, then all items linked to the item corresponding to the hash code value are checked for similarity.

As mentioned before, when search operations are performed, *equals* method allows fuzzy string matching.

The approximate matching algorithm use by default is Levenshtein Distance [8] algorithm. This algorithm calculates the distance between two words. The distance represents the number of modifications (insert / update / delete) needed to make the words equal. The matching performance is flexible and can be controlled by making *equals* method use a custom maximum distance threshold value. By default this value is 2 and it was decided after empirical results.

Another approximate matching algorithm used is Hamming [9] distance, which can compare equal length string only, and represents the number of positions where the corresponding symbols are different.

III. Examples

To illustrate the way FHM works let's consider two examples.

A. Fuzzy Law Terminology Dictionary

We want to identify, in plain text, law specific terms. The terms have to be identified even though they are not in the canonical form. For this we will use a FHM to build a law specific terminology dictionary. We will use that terminology dictionary to recognize specialized law terms in the given text.

The first step is creating and populating the terminology dictionary. The FHM used will have the default settings. Fig. 2. presents the process of populating the FHM with law terminology data. For this case the *substring(0,4)* pre-hashing function was chosen. Let's consider adding term "*action*" with the associated interpretation into the FHM. The pre-hashing function returns "*acti*", and the hashing function has calculated value 12 for this string. Since there is no other entry stored on this location, this entry is saved here.

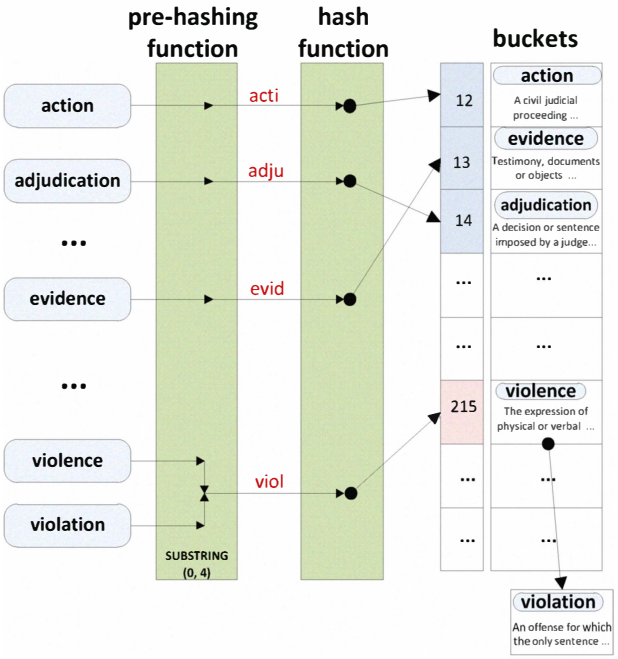

Fig. 2. Populating FuzzyHashMap with Law Terminology data

Now let's jump to the "*violation*" dictionary entry. Notice that in this case the pre-hashing function returned "*viol*" the same as it returned for the "*violence*" entry. The hashing function will calculate the same value for both entries, *215*. In this case we have a collision, so the hash map will call *equals* method to check whether "*violation*" is already saved in the map. After performing this checking, with negative result, the map knows it's a different entry, so it creates a new location for the new entry, which will be linked to the location of "*violence*" entry.

Let's put the fuzzy dictionary to work now. So we consider we are parsing the following phrase:

"the *judge* has the option of either *adjudicating* you as guilty or.."

Each word is checked against the dictionary. When arriving to "*adjudicating*" term, as presented in Fig. 3, the dictionary will search by firstly pre-hashing the term. The hash code for the resulted string "*adju*" is computed, and it points to the location 14, where "*adjudication*" entry is stored. The *equals* method is called to check the term against the entry at position 14. *Equals* method now uses approximate matching. The Levenshtein distance (which is the default approximate matching algorithm in FHM) between "*adjudication*" and "*adjudicating*" is 2. While by default FHM has threshold value 2, the *equals* method will return *true*. So the word *adjudicating* has been associated to the term *adjudication* from the dictionary.

In conclusion, the FHM enables finding terms that are not in their canonical form, in a very efficient way. To make this possible, this is error tolerant, so it may do mistakes, but a good threshold and algorithm tuning improves the performance of the FHM.

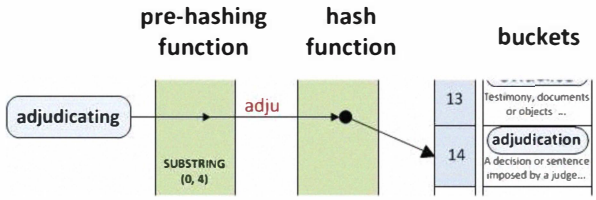

Fig. 3. Fuzzy searching for word "*adjudicating*"

B. *Phone book*

The phone book is storing contacts (name of a person and the phone number). The example is very similar to the precedent one, juts that here the pre-hashing algorithm used is soundex algorithm. Manipulating data in the phone book is presented in Fig. 4.

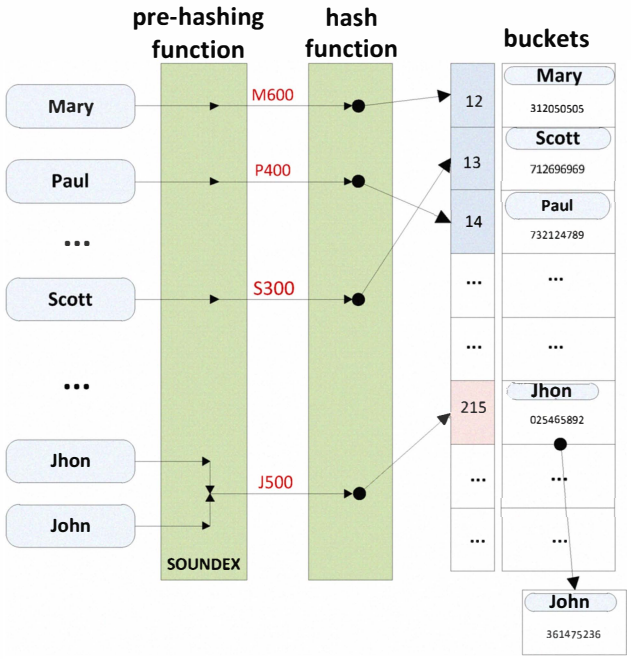

Fig. 4. Fuzzy Phone Book

The advantage of using soundex algorithm is that it helps on having a better words clustering, according to the way they sound. So this fuzzy hash book could be used to enable users find contacts even if they are misspelling the name of a contact person, or when using speech recognition, which is error prone, to select a contact person. The detailed procedure of adding and searching entries in the FHM will not be presented because it is similar to the previous example.

Additionally to the described examples, some other use case scenarios for FHM data structure were identified:

- *Natural language processing*: Here the fuzzy words recognition can bring language independence for tasks like lemmatization or part of speech tagging (POS).
- Information Retrieval
- *Speech Recognition*: fuzziness may help the recognition performance
- Optical Character Recognition
- *"On-the-fly" type-ahead search:* fuzzy HashMaps could also be an alternative to the current work done in this area for improving database query performance [10]
- Spelling correction
- Machine translation

IV. Test

Two main test types were used to measure the performance of the FHM., one for accuracy and another one for time consumption.

A. *Accuracy*

The first one was testing the accuracy of fuzzy matching. However, testing the accuracy of fuzzy matching is not very relevant, because the results can change dramatically form one case to another, depending to the *text used in the test*, the *size of the map* and the *settings* of the FHM. For this type of test, the settings used for the FHM were:

- Substring(0,4) hashing function
- Levenshtein Distance fuzzy matching algorithm
- Distance threshold value 2

The test strategy was to use the FHM as a medical terminology dictionary. The FHM medical dictionary was populated with 1030 English medical terms.

Using text from the *American Family Physicians Journal*, an application has been developed to parse this text and to try to identify the medical terms by searching fuzzy matches in the dictionary for each word in the text.

Here are some test results: for a text having a total number of 568 words, 43 words have been identified as medical terms. From these 43 words 9 were incorrect matches. This means an approximate 80% accuracy for the fuzzy matching.

In another test done on text from an eMedicine web site, containing 2730 words, 260 were recognized, from which, 7 were incorrect matches. This means 97% accuracy.

As mentioned before, these values can change for different test conditions. In our case the biggest improvement can be done by populating the map with more medical terms.

B. *Time consumption*

The second test type was done to analyze the speed of the FHM. The speed is compared to HashMap speed, tested in the same conditions. A custom optimized TreeMap was tested too, but the results are not listed because they were very similar to the results of the FHM, so the graphic would make no difference. For these tests each map was populated with 30000 words (random strings). Than 1000 words were searched in the maps, in three tests:

a. All the words that were identified as match were exact match. Fig. 5 illustrates the results; the horizontal axis represents the number of matched words from the 1000 words searched. This test results show that FHM has an approximate constant time of 5 milliseconds regardless the number of matching words. The regular HashMap, as expected, increases its time performance while the number of exact matches grows.

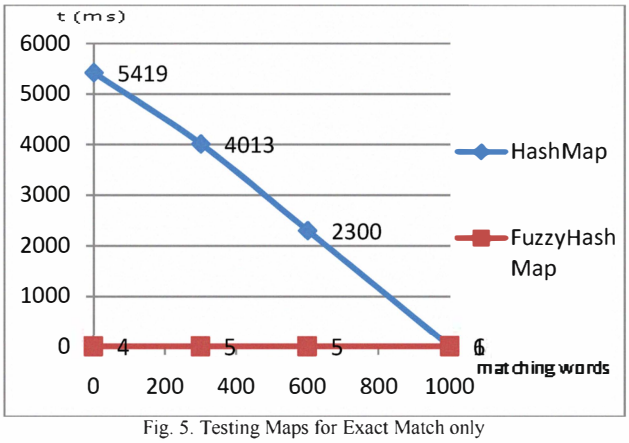

Fig. 5. Testing Maps for Exact Match only

When all the 1000 searched words are exact match, HashMap has the best performance, 1 millisecond. However, when the number of exact matches is small, the performance of HashMap is very bad compared to FHM.

b. In this test all the words identified as match were approximate matches only. Here (Fig. 6) we see the performance of HashMap is bad and not increasing even if all the words are matched.

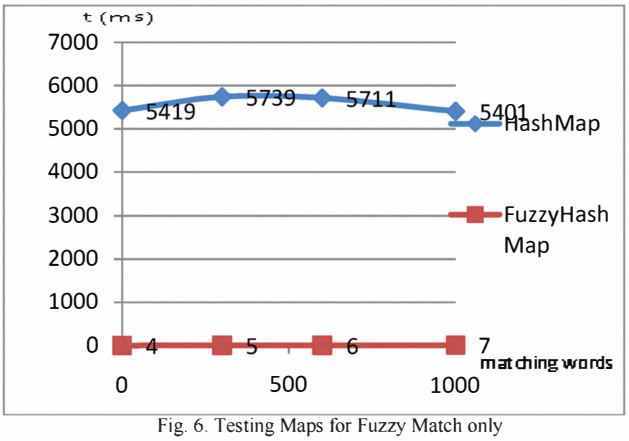

Fig. 6. Testing Maps for Fuzzy Match only

c. In this test the words identified as match were 50% exact matches and 50% fuzzy matches.

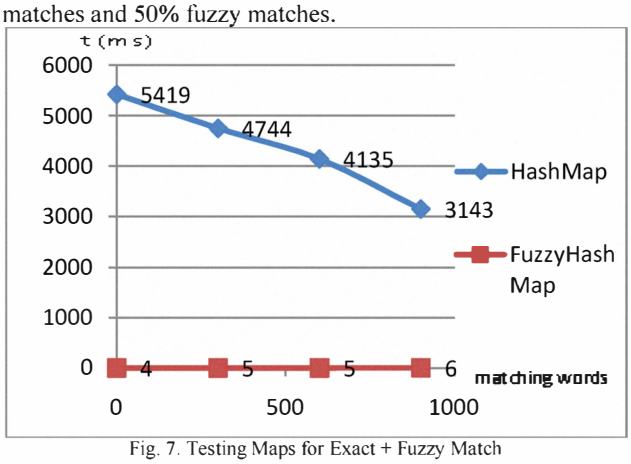

Fig. 7. Testing Maps for Exact + Fuzzy Match

For this case, in Fig. 7. one can see the performance of HashMap is increasing proportionally with the number of matched words.Once again we can notice that FHM time consumption is almost constant for this test too.

From all these tests the following conclusions arise:

- FuzzyHashMap do not manifest a big performance variation proportional to the number of matched words (exact and fuzzy)
- The performance of the FHM is almost equal to the performance of TreeMap, but has better memory consumption and allows more fuzzy string operations.
- Regular HashMap has better performance than FHM when all the searched words are exact matched, but it has very bad performance when the number of exact matched words decreases. For approximate matched words the performance of the HashMap is bad and independent of the number of matched words.

IV. Conclusion

This article describes an extension to the regular java HashMap generically called *FuzzyHashMap*, trying to cope with the main disadvantage of the conventional hash map: the poor performances when working in uncertain conditions. The article is more oriented on empirical research. However the theoretical foundation for the presented data structure is described.

The real case scenario presented in this paper is part of a research project, dealing with interpretation of natural language and specialized terminology.

FuzzyHashMap data structures proved to have good performance on working with uncertain data. A big advantage is its flexible and the fact that the developer can use different algorithms for pre-hashing and fuzzy matching. This allows the achievement of better performance for different use scenarios. Also time performance, compared to the use of regular HashMap for the same task, is much better. The data structure presented in this paper is available as open source, so that and anyone can use it. This will probably help at improving the performance of FHM, with support from the community.

As future work, integration of more string metrics algorithms are planned, together with finding the most efficient algorithms and fine tuning settings. A collection of string metrics algorithms available as open source [11] has already been identified for being included in the hash map. Another future work, determined by the good results the custom TreeMap data structure had in the conducted tests, will be to extend and optimize this data structure for fuzzy enabled operations.

As final conclusion, the presented data structure enables fuzzy like operations on top of the basic available functionality. Being in the area of soft computing the data structure is tolerant to imprecision and partial truth, but has proven very good performance for fuzzy like operations.

REFERENCES

[1] http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashMap.html.

[2] C. Galvez, F. Moya-Anegón; Approximate Personal Name-Matching Through Finite-State Graphs; Journal of the American Society for Information, 2007

[3] S Ji, G Li, C Li, J Feng; Efficient type-ahead search on relational data: a TASTIER approach; Proceedings of the 35th SIGMOD international conference on Management of data, 2009

[4] M Hadjieleftheriou, C Li; Efficient approximate search on string collections; Proceedings of the VLDB Endowment, 2009

[5] A. Urrutia; L. Tineo; C. Gonzalez; FSQL and SQLf: Towards a Standard in Fuzzy Databases; Handbook of Research on Fuzzy Information Processing in Databases; Pages: 270-298; 2008

[6] FuzzyHashMap open source project web address: http://fuzzyhashmap.sourceforge.net/

[7] R. C. Russell and M. K. Odell; Soundex algorithm patent

[8] V. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. Probl. Inf. Transmission 1, 8-17. 1965

[9] R. W. Hamming; Error Detectiong and Error Correcting Codes; The Bell System Technical Journal, vol xxIx, No. 2; 1950

[10] Guoliang Li, Shengyue Ji, Chen Li, Jianhua Feng; Efficient type-ahead search on relational data: a TASTIER approach; International Conference on Management of Data, Rhode Island, USA, 2009

[11] SimMetrics: http://www.dcs.shef.ac.uk/~sam/stringmetrics.html