Scientific Journal of Impact Factor(SJIF): 3.134

e-ISSN(O): 2348-4470
p-ISSN(P): 2348-6406

International Journal of Advance Engineering and Research Development

Volume 2,Issue 5, May -2015

OPTIMAL SOLUTION TO FIND IDENTICAL LINES

Tapasya Dinkar

Computer And Science Department,Saffrony Institute of Technology ,Linch,Mehsana

Abstract: *In this paper we are presenting some solutions on how to find identical lines in a file when the lines are so long that they cannot fit in a memory at the same time. Suppose that There are 100000 lines in a file. Out of these only 2 lines are identical. Rests are unique. Each line is so long that it cannot fit in memory. We tried to find an optimal solution to find identical lines.*

Keywords: *Grep command, Hash function, Hash table.*

I. INTRODUCTION

String matching is a classical problem of computer science. The basic task is to find all the occurrences of a pattern string in a text string, where both of the strings are drawn from the same alphabet. Here we are given a file which contains 100000 lines, which is a large file. And in this file only two lines are identical. And the lines are so big that it is not fitting in the memory. As we know that to execute anything it should be first fit to memory then only it can implemented. So here first we have to find the technique to fit that long line into the memory and then some other methods to find that two identical lines from the file.

1.1 Grep command in linux:

Grep is a command which is basically used in linux system.Grep searches the input files

for lines containing a match to a given pattern list. When it finds a match in a line, it copies the line to standard output (by default), or whatever other sort of output you have requested with options.

Though grep expects to do the matching on text, it has no limits on input line length other than
available memory, and it can match arbitrary characters within a line. If the final byte of an input file is not a *newline*, grep silently supplies one. Since newline is also a separator for the list of patterns, there is no way to match newline characters in a text.

Name:
grep, egrep, fgrep print lines matching a pattern

Synopsis:

grep [*options*] *PATTERN* [*FILE...*]
grep [*options*] [-e *PATTERN* |- f *FILE*] [*FILE...*]

Description:

Grep searches the named input *FILE*s (or standard input if no files are named, or the file name is given) for lines containing a match to the given *PATTERN*. By default, grep prints the matching lines. In addition, two variant programs egrep and fgrep are available. Egrep is the same as grep E. Fgrep is the same as grep F.

Some options are:

-x, --line-regexp
Select only those matches that exactly match the whole line.

-w, --word-regexp
Select only those lines containing matches that form whole words. The test is that the matching substring must either be at the beginning of the line, or preceded by a non-word constituent character. Similarly, it must be either at the end of the line or followed by a non-word constituent character. Word constituent characters are letters, digits, and the underscore.

-n, --line-number
Prefix each line of output with the line number within its input file.

There are many more options in Grep command.

1.2 REGULAR EXPRESSIONS

A regular expression is a pattern that describes a set of strings. Regular expressions are constructed analogously to arithmetic expressions, by using various operators to combine smaller expressions.
Grep understands two different versions of regular expression syntax: ``basic'' and ``extended.'' In GNU grep, there is no difference in available functionality using either syntax. In other implementations, basic regular expressions are less powerful. The following description applies to extended regular expressions; differences for basic regular expressions are summarized afterwards.
The fundamental building blocks are the regular expressions that match a single character. Most characters, including all letters and digits, are regular expressions that match themselves. Any metacharacter with special meaning may be quoted by preceding it with a backslash

Basic plan for GREP by using DFA:
• build DFA from RE
• simulate DFA with text as input

No backup in a DFA
Linear-time because each step is just a state change

The DFA can be exponentially large (can't afford to build it).

Basic plan for GREP (revised) by using NFA:
• build NFA from RE
• simulate NFA with text as input
• give up on linear-time guarantee

Limitations:

Although Grep command is used to find the identical line it has a limitation too. It only match tohose line which fits in the memory. If the line is not stored in the memory because it is too long the grep command is not work.

II. SUGGESTED METHOD

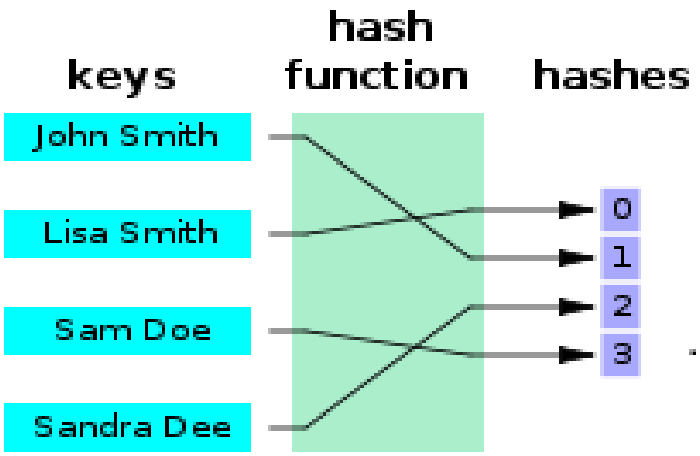
1. As we know that to execute any program or implement any method that program should be in the main memory then only it can run to completion. Here we have to put a long line in a memory. The main memory is of limited size.
The whole line is not stored in the memory so we can divide the line into number of parts so that it can easily be fitted in the memory .So that it will look like one pattern stored in the memory.
As it is given grep command is use to find the identical lines using NFA we can a search the identical lines using this technique also.

2.Also we can divide the line into two part and the first compare the first half with the lines and then select only those lines for next part comparison which are identical with the first part of line. Here the number of searching the line in second part is reduce.
But there is a drawback of this method that if only some character of line is differ from the first part then the time taken to search the identical lines is increases.

3.When storing lines in a large unsorted file, one may use a hash function to map each line to an index into a table *T*, and collect in each bucket *T*[*i*] a list of the numbers of all lines with the same hash value *i*. Once the table is complete, any two duplicate records will end up in the same bucket. The duplicates can then be found by scanning every bucket *T*[*i*] which contains two or more members, fetching those lines, and comparing them. With a table of appropriate size, this method is likely to be much faster than any alternative approach (such as sorting the file and comparing all consecutive pairs).
We can use the hash function. As we calculate the hash value of each lines using hash function and the store the hash value of each line in bucket. And compare one by one if we get two hash values same the we get the two identical lines.
And here we get 100000 hash values to compare.

2.1 Hash function:

A hash function is any function that can be used to map digital data of arbitrary size to digital data of fixed size, with slight differences in input data producing very big differences in output data. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes. One practical use is a data structure called a hash table, widely used in computer software for rapid data lookup. Hash functions accelerate table or database lookup by detecting duplicated records in a large file.



2.2 Hash table:

Hash functions are primarily used in hash tables, to quickly locate a data record (e.g., a dictionary definition) given its search key (the headword). Specifically, the hash function is used to map the search key to an index; the index gives the place in the hash table where the corresponding record should be stored. Hash tables, in turn, are used to implement associative arrays and dynamic sets.
Typically, the domain of a hash function (the set of possible keys) is larger than its range (the number of different table indexes), and so it will map several different keys to the same index. Therefore, each slot of a hash table is associated with (implicitly or explicitly) a set of records, rather than a single record. For this reason, each slot of a hash table is often called a *bucket*, and hash values are also called *bucket indices*.
Thus, the hash function only hints at the record's location — it tells where one should start looking for it. Still, in a half full table, a good hash function will typically narrow the search down to only one or two entries.

2.3 There are some algorithms to find the hash function:

For most types of hashing functions the choice of the function depends strongly on the nature of the input data, and their probability distribution in the intended application.

2.3.1 Trivial hash function

If the datum to be hashed is small enough, one can use the datum itself (reinterpreted as an integer) as the hashed value. The cost of computing this "trivial" (identity) hash function is effectively zero. This hash function is perfect, as it maps each input to a distinct hash value.
The meaning of "small enough" depends on the size of the type that is used as the hashed value. For example, in Java, the hash code is a 32bit integer. Thus the 32bit integer Integer and 32bit floating point Float objects can simply use the value directly; whereas the 64bit integer Long and 64bit Floating point Double cannot use this method.
Other types of data can also use this perfect hashing scheme. For example, when mapping character strings between upper and lower case, one can use the binary encoding of each character, interpreted as an integer, to index a table that gives the alternative form of that character ("A" for "a", "8" for "8", etc.). If each character is stored in 8 bits (as in ASCII or ISO Latin 1), the table has only 28 = 256 entries; in the case of Unicode characters, the table would have 17×216 = 1114112 entries.
The same technique can be used to map two letter country codes like "us" or "za" to country names (262=676 table entries), 5digit zip codes like 13083 to city names (100000 entries), etc. Invalid data values (such as the country code "xx" or the zip code 00000) may be left undefined in the table, or mapped to some appropriate "null" value.

2.3.2 Perfect hashing

A hash function that is injective—that is, maps each valid input to a different hash value—is said to be perfect. With such a function one can directly locate the desired entry in a hash table, without any additional searching.

2.3.3 Minimal perfect hashing

A perfect hash function for n keys is said to be minimal if its range consists of n *consecutive* integers, usually from 0 to $n-1$. Besides providing single step lookup, a minimal perfect hash function also yields a compact hash table, without any vacant slots. Minimal perfect hash functions are much harder to find than perfect ones with a wider range.

2.3.4 Hashing uniformly distributed data

If the inputs are bounded length strings and each input may independently occur with uniform probability (such as telephone numbers, car license plates, invoice numbers, etc.), then a hash function needs to map roughly the same number of inputs to each hash value. For instance, suppose that each input is an integer z in the range 0 to $N-1$, and the output must be an integer h in the range 0 to $n-1$, where N is much larger than n. Then the hash function could be $h = z \bmod n$ (the remainder of z divided by n), or $h = (z \times n) \div N$ (the value z scaled down by n/N and truncated to an integer), or many other formulas.
$h = z \bmod n$ was used in many of the original random number generators, but was found to have a number of issues. One of which is that as n approaches N, this function becomes less and less uniform.

2.3.5 Rolling hash

In some applications, such as substring search, one must compute a hash function h for every k character substring of a given n character string t; where k is a fixed integer, and n is k. The straightforward solution, which is to extract every such substring s of t and compute $h(s)$ separately, requires a number of operations proportional to $k \cdot n$. However, with the proper choice of h, one can use the technique of rolling hash to compute all those hashes with an effort proportional to $k + n$.

REFERENCES:

[1] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. An optimal algorithm for generating minimal perfect hash functions. Information Processing Letters, 43:257–264, 1992.
[2] J. Levandoski, E. Sommer, and M. Strait, "Application Layer Packet Classifier for Linux." http://l7-filter.sourceforge.net/.
[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. Introduction to Algorithms. MIT Press, 1990.

