HMM-Based Selectivity Estimation for SQL LIKE Queries

Abstract: Selectivity estimation has crucial importance for database management systems, particularly in string databases where queries with wildcards are prevalent. This paper introduces a novel approach to query optimization by addressing the challenge of selectivity estimation for string queries with wildcards using Hidden Markov Models (HMMs). While previous algorithms have been developed for this purpose, their effectiveness on large datasets and complex query patterns has been limited. Algorithms are proposed based on the statistical analysis of the strings in the dataset. The proposed algorithms aim to optimize the state transition and observation probabilities of HMMs by mapping characters with indexes. However, experimentation reveals that while the algorithm may be accurate for small datasets, it falls short on large datasets and queries of the form $\%w_1\%w_2\%$ or $\%w_1\%w_2\%...\%w_n\%$. Despite these limitations, this approach opens new doors for future research and development, offering a fresh perspective on selectivity predicates and paving the way for innovative solutions. Improvements are needed to enhance performance, but this algorithm represents a promising avenue in the field of query optimization.

1. Introduction

Query optimization is an important process in database systems. In most cases, estimating the correct selectivity values to run appropriate algorithms are of great importance for proper optimization. In this paper, we will try to find a solution to the string selectivity prediction problem specifically for SQL LIKE queries.

String selectivity prediction can be made using a number of methods nowadays. For example, accurate values can be found by using full-text pattern matching, such as the Naïve Algorithm and the Knuth-Morris-Pratt Algorithm [1], or trie-based methods. However, full-text search methods work inefficiently on large data sets, as it is necessary to rescan the entire database for each query and trie requires huge memory. Our aim is to develop an algorithm to find the selectivity of the strings without scanning the entire database. Differently, we will propose some methods by statistically analyzing words and using Markov Chains and Hidden Markov Models (HMM). Since we could not find a paper in which Markov Chains were used to find string selectivity prediction before, it will be a first in its field and perhaps will open the door to new research methods.

Our primary goal is to obtain the result without scanning the entire rows and as quickly as possible, typically by performing a process as many times as necessary. Since we will use statistical values, future deviations in these values can be predicted. Therefore, our aim will be to calculate an average value for all queries rather than calculating an accurate value. While doing this, the most important factor will be to find the right optimization values for our states. We will

try to find more optimized values by proposing a new algorithm by considering statistical data such as the number of letters in certain indexes, and the algorithm we will use in the calculation part of the query selectivities is the Forward Algorithm [2], thanks to its easy applicability.

Markov chains focus on probability and predict the next state using the properties of the current state [3]. The biggest problem in the algorithms we are going to propose is that we choose the states over the letter indexes or letters in the alphabet, and the next state does not actually depend only on the current state. However, despite this, we aim to overcome this problem by using different calculation methods and finding an average value quickly as we have emphasized. Again, in this method, as in other algorithms, the biggest problem is finding different combinations caused by the wildcard symbol. Instead of trying different combinations one by one, we plan to solve them by adding the wildcard symbol to our alphabet.

In the following sections, we will review the studies on this subject in the past, explain our own methods, and finally share the results of our experiments.

2. Related Work

The paper [4] is the first to consider selectivity prediction in SQL LIKE queries. It considers Suffix Trees to predict the selectivity of the SQL LIKE queries in the format $\%s\%$ where s might be one or more characters. Suffix Tree is kind of a Trie with only one difference. It only holds the suffixes of the strings. It normally works by pushing all the suffixes of the strings to the Suffix Tree and holding their counts. By doing that, any kind of string search in the format $\%s\%$ can be calculated easily. However, database systems should be fast and memory efficient and this method uses so much memory. To decrease this, the Pruned-Suffix Tree is invented. If the count of any node is less than some threshold then that node is pruned. It helps to make code memory efficient while making it harder to predict selectivity. Because the selectivity query might be pruned from the tree and how should we estimate the value of that node? They used Independence-based, Child Estimation-based, and Depth Estimation-based strategies to predict the selectivity in this case. The Independence-based strategies assume that all nonintersecting substrings are independent of each other and the result can be found by multiplying the results of substrings. Child Estimation-based strategies focus on calculating approximate selectivity for pruned children and Depth Estimation-based strategies predict average values based on the level of the nodes. The evaluation of the experiments was made by considering positive single, positive double, and negative string patterns. According to the experiments, most of the time, the best performers of the strategies are the Independence-based ones, and Child Estimation-based strategies follow them. In general, the results can be considered good enough since they are a starting point in the area. One of the drawbacks of the paper is it only focuses on selectivity prediction for $\%s\%$ queries and offers a similar solution to Independence-based strategies for the queries in the format $\%s\%t\%$, however, the relative error is significantly increasing for that type

of query. The other drawback is it does not consider formats $s\%$ or $\%s$ and their algorithms are not properly work in these formats. However, these all can be forgivable since it is the first in its area.

The paper [5] focuses on improving the selectivity estimation algorithm for $\%s\%$ format strings that is proposed by [4]. The main focus of the paper is to criticize the KVI algorithm for using complete conditional independence methodology, and it offers a new approach, conditional dependence as the maximal overlap between consecutive substrings. For example, to estimate the selectivity of 'jone', they use 'jon' and 'one' as substrings and their maximal overlap as 'on'. The proposed algorithm is named as Maximal Overlap (MO) algorithm and it is based on the short memory property. It uses Markov Chains probability assumption by the perspective of using the previous substring to estimate the next substring selectivity. The experiments that are held by the authors are shown that the MO algorithm is better than KVI. The paper also categorizes suffix trees as p- and o- suffix trees, where p- stands for presence, and o- stands for occurrence. o- suffix trees count all the substrings within one string, whereas p- suffix trees count the substrings only one time per string to state whether it occurs or not. Using o- suffix trees is more advantageous to estimate the results of selectivities better. Besides MO, two algorithms, MOC and MOLC, are proposed to increase accuracy. Maximal Overlap with Constraints, MOC, is using left and right ancestor substrings and their children's frequency in the pruned suffix tree, and determines negative string queries more accurately. In this algorithm, while checking the selectivity of 'jon', it considers checking the child nodes of 'jo', if, for example, the frequency of 'jo' is 20, 'joe' is 10, and 'jok' is 10, it is obvious that there is no 'jon' in the pruned suffix tree, and in the dataset. This approach increases the performance of negative string queries. Maximal Overlap on Lattice with Constraints, MOLC, creates a lattice from the left and right parent substrings of the query string and checks all the node frequencies with constraints as in MOC. It estimates all child node frequencies from its parent's frequencies and creates a similar shape to a lattice. The accuracy decreases as MOLC, MOC, MO, and KVI, whereas the algorithmic complexity increases in a reverse way. The proposed algorithms are a new way of thinking about substring selectivity estimation, however, it only tries to optimize the first proposed algorithm, and it is unclear that the proposed algorithms always perform better than KVI, since there is no mathematical proof of this, and they are all based on assumptions.

The paper [6] tries to solve overcome the underestimation problem of the selectivity estimation of SQL LIKE queries. The algorithm is based on Short Identifying Substring Hypothesis. If the selectivity of $s'$is not larger than $(1 + e)\ *\ selectivity\ of\ s$, and the length of s' is smaller than the length of $s\ *\ b$, where e and b are between 0 and 1, then we can say that s is a short identifying substring of s'. Regression-Trees and Q-gram tables are used to find possible candidate substrings for a given string. First, 3-gram tables are generated to store frequencies of all 3-length substrings. Then, 4, 5, and $|L|$ length substrings are generated using these 3-length substrings, where L is the length of the query string. The Markov and QG Estimators are used to

generating the substrings. The Markov Estimator correlates based on the dependence of strings and substrings, i.e the $ME - selectivity\ of$ '$\%novel\%$' $= P(nov) \cdot P(ove \mid nov) \cdot P(vel \mid ove) = f(nov)/N \cdot f(ove)/f(ov) \cdot f(vel)/f(ve)$, where the QG Estimator limits the upper bound of a string as the minimum selectivity of substrings, which can be found from the Q-gram Tables. The smallest Markov Estimation selectivity in each level is chosen in the Regression-Tree to find identified substrings in each level of the Regression Tree. The selectivities of these strings are used in the Regression-Tree Combination function which is the weighted geometric mean of these different selectivity values. This approach helps us to find accurate results due to the non-linear dependence between substrings. The formula for this function is $exp(w1 * log(ME - Selectivity(s1)) + w2 * log(ME - Selectivity(s2))) + ... + wL * log(ME - Selectivity(sl))$ where si's are the substrings of the string s, and wi constants are some constants are estimated or learned with the help of machine learning algorithms. The paper also compares its results for range queries. In the end, the CRT Algorithm performs better than QG and Markov Estimators for positive range and string queries. However, the Markov Estimator performs better for negative string queries.

In the paper [7], the under- and over-estimation problems of KVI and MO Algorithms are focused on. They suggest modifying the Pruned-Suffix Tree which is proposed by [4]. The modified Pruned-Suffix Tree, RPST, has two new rules: The second level of the tree cannot be pruned and the first characters of the Pruned nodes are stored in the Pruned-Nodes Table. First of all, the accuracy of predicting the selectivity of negative queries will be precise with the prevention of pruning the second level, and the selectivities will be calculated more accurately using PNT. For example, if a string occurs in the database, it can not be predicted as a negative string since its result will be greater than zero. Secondly, the accuracy of selectivity estimations is increased by checking whether strings exist in the PNT or not. The algorithms EKVI and EMO are proposed in the paper to increase selectivity estimation precision, and according to the experiments, EKVI works better than KVI and EMO works better than MO algorithms. Specifically, this approach helps to decrease severe under- or over-estimations. In addition, when the disk capacity grows the difference between errors is getting significant, and EMO works more performant than others. The paper focuses on decreasing severe under- and over-estimations, however, besides their more accurate estimations, it still uses the algorithms of KVI and MO. These algorithms have drawbacks since KVI focuses on the independence of substrings and MO focuses on the dependence on previous substrings. Both approaches do not have precise results yet, and more research should be done to improve their accuracy.

The paper [8] focuses on two proposed algorithms named MOF and LBS. The Most Frequent Minimal Base Substring, MOF, finds all the minimal base substrings, where a minimal base substring is defined as minimal if it does not have any other base substrings. It uses edit distance to find all the substrings to find the base substrings. Edit distance is to minimum required operations to reach from one string to another by performing Insert, Delete, and Replace

operations. ? character is added to the alphabet to consider Replace operations, and N-gram tables are used to store all the base substrings. In addition to the MOF algorithm, the Lower Bound Estimation (LBS) algorithm is the extended version of the MOF algorithm. The MOF algorithm only checks the most frequent minimal base substring, whereas the LBS algorithm checks all the minimal base substrings and estimates using the technique called set resemblance. In this algorithm, Monte Carlo Set Hashing is used to calculate set resemblance. The proposed algorithms compared with the S-SPEIA, a method based on clustering, and RS, a method with no space overhead, algorithms in terms of error rate, pre-computation time, query processing time, and space need. The paper also considers positive queries in the format $\%s\%$, and $\%s1\%s2\%s3\%\ldots\%$, and also negative queries. In most cases, MOF and LBS algorithm performs significantly better, and LBS has more accurate estimations than MOF. If there is enough space, LBS is a preferable choice rather than MOF. The biggest drawback of the algorithms is the pre-computation time, it might be improved rather than compared algorithms, however, it still requires a considerable amount of time to use.

The paper [9] focuses on applying sequence pattern mining techniques for selectivity estimations. Sequence pattern mining is first proposed to examine customer operations in large databases, i.e advertisement purposes, by Agrawal and Srikant. Sequence pattern mining aims to find which items come after each other, and where items might be next to each other or not. Application to strings of this technique is meaningful since strings also contain patterns. The algorithm, proposed by the paper, focuses on mining sequential patterns and then finding frequent closed ones to eliminate strings and increase memory efficiency. After patterns are found, patterns are bucketed and buckets are stored in a table, which is histograms. For the selectivity estimation part, a new string is checked whether it has an exact match, an encapsulated match, or no match. In the exact match case, the selectivity can be determined directly, and the average of the frequencies is calculated in the encapsulated match case. For the no-match case, selectivity is found with the help of the minimum support threshold, which is determined initially for pattern matching. The most advantageous part of the algorithm is the support of the queries in $\%s1\%s2\%s3\%s4\%$ formats, where si can be any string. These advantages can easily be seen in the experiment results. When compared with the state-of-the-art technique, the sequential pattern matching technique, LBS, works with significantly less relative error on positive queries in formats $\%s1\%s2\%$ and $\%s1\%s2\%s3\%s4\%$...%. Besides that SPH requires four times more memory than LBS in the building phase, and the time and memory requirements are 7 and 120 times better than LBS in the query optimization phase. The most important drawback of the algorithm is the insufficiency of estimation on sequential positioning within frequent patterns, i.e whether A is next to B or not in the ABB frequent pattern. In general, the proposed algorithm is a huge advancement in the selectivity prediction of SQL LIKE queries.

3. Methods

3.1. Single-Letter HMMs

This section contains our main algorithm which is predicting the selectivities using Hidden Markov Models [2]. However, we did not use the HMMs directly, instead, we override the Hidden Markov Models a little bit rather than using them directly. The main idea of the algorithm starts with mapping the letters with their indexes. In HMM, there are states, state transitions, and also observations for each state. In the basic case, the state transitions will be linear where only the consecutive transitions will be connected, and the observation values will be the probability of letters being in the specified index. As you can suggest, the state transition probability between sequential states will be %100, since there will be only one outgoing transition from each state. An example construction of the HMM networks for word set {“ahmet”, “ali”} can be seen in Figure 1. In the figure, the circles represent the states and the rectangles represent the observation values for each specific index. The states go from 1 to 5, and

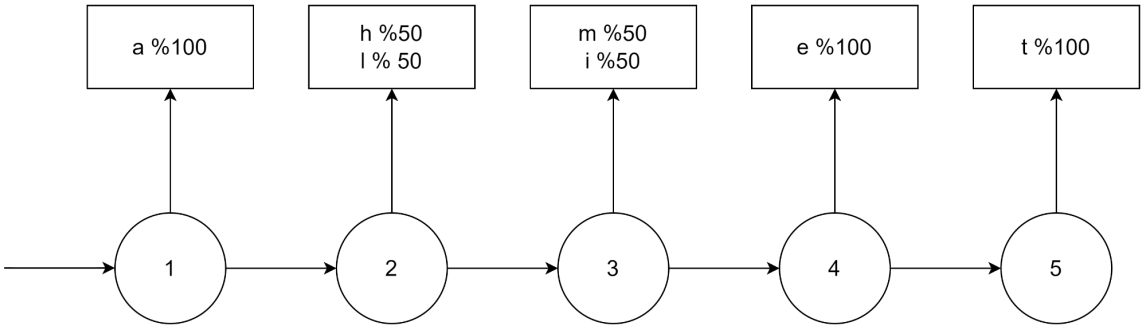

Figure 1: Example Single-Letter HMM

the observations contain the probabSilities for the relevant letters. For example, the probability of ‘i’ being in the 3rd index is %50 or ‘t’ being in the 5th index is %100. This HMM can be constructed from scratch or it can be updated incrementally as the words are changed, inserted, deleted, or updated. To enable this, we changed the structure of observations as in Figure 2. Instead of directly keeping the probabilities in observations, the frequency of the letters and total letter value for each state will be held in the observations tables. The total letter for each state is added to the table using the percent (%) symbol, which is used as a wildcard symbol. The basic formula $freq\ of\ letter/\ freq\ of\ wildcard$ can be used to calculate each letters observation probability while allowing update operations easily. The update operations can be done by $freq\ of\ x_i\ +=\ 1\ and\ freq\ of\ wildcard\ +=\ 1$ for ith state where the ith letter of x is x_i. Until now, everything can be easily applicable to HMMs. However, the HMM in Figure 1 can only be used in exact query matching cases, i.e. query “ahmet” or

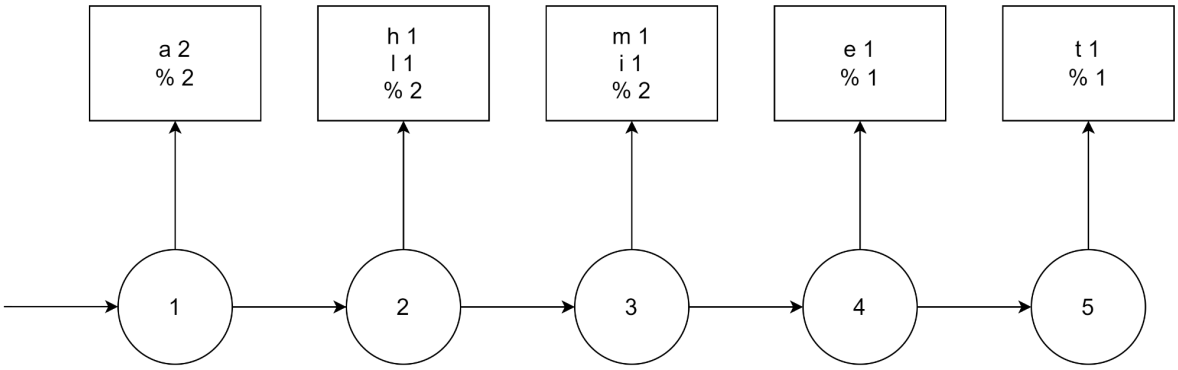

Figure 2: Example Extended Single-Letter HMM

“alimet”, but not for “ah%et” where % represents zero or more characters. The state transitions are also evolved for that purpose as in Figure 2. However, the HMM in Figure 2 can be used for queries with wildcards, or in other words, SQL LIKE queries. The basic idea is to check the sum of the selectivities of all possible cases for that type of query, i.e. “aht”, “ah_t” and “ah__t” for the query “ah%t”, where the underscore character represents only one character. The observation of the underscore can be estimated as %100 if there is any observation for that state, or as %0 if there is no observation. The reason is the sum of probabilities in observations should be %100, and the underscore can be any character. A recursive function should be implemented to check

```
function get_probability_helper(word, branches, word_index, state_index):
    if word_index equals word.size():
        return 1.0

    prob = 1.0
    if branches[word_index] is false:
        prob *= states[state_index][word[word_index]] / states[state_index]['%']
        return prob * get_probability_helper(word, branches, word_index + 1, state_index + 1)
    else:
        prob *= states[state_index][word[word_index]] / states[state_index]['%']
        sub_prob = 0.0
        remaining_letters = word.size() - 1 - word_index
        for i from state_index + 1 to states.size() - remaining_letters:
            sub_prob += get_probability_helper(word, branches, word_index + 1, i)
        return prob * sub_prob
```

Algorithm 1: Recursive Query Function

all the possible cases easily. The pseudocode of the function can be seen in Algorithm 1. The branches array holds whether there is a ‘%’ character after the character or not for each letter. If yes, branching should occur as in Figure 2. Branching means that instead of only transiting from ith state to $i + 1$th state, all other states also should be checked. The sum of these branching probabilities might be summed at the end of the function. In a real-life scenario, zero or more characters are inserted instead of a ‘%’ character, but in this approach, there is no need to calculate every possibility. Because we are sure that all the probability values for ‘%’ are already %100 which means that in a certain index, there is a following character from any character to

any other character, and if not, the probability should be %0. The algorithm guarantees that if there is no such transition, the probability will be zeroed somehow. Branching helps us to bypass these unnecessary steps, and focus on calculating the results in an efficient manner. Giving 100% percent probability to these state transitions is done also to keep the calculations meaningful, which leads to having 100% for every transition probability.

In addition to all these things, to make everything easier the '$' character is added at the beginning of every word set and query word to make the recursive query function implementation easier for query words starting with '%', i.e. '%et'.

Even though the construction and query methodologies of the Extended Single-Letter HMM seem logical, it generally underestimates exact queries. Imagine the case of the word set {"ahmet", "furkan"}, and we are searching for the query "ahmet". The result of the selectivity calculation for this query will be $(1/2)$ * $(1/2)$ * $(1/2)$ * $(1/2)$ * $(1/2)$ = $1 / 32$ which is $1/2$ in the actual case. The main reason is our calculation methodology only depends on the current letter for indexes. However, it should be also dependent on letter transitions, in a detailed way: which letters come after which letters. The next proposed algorithm will be the overridden version of this approach.

3.2. Extended Double-Letter HMM

In the Extended Double-Letter HMM version, the observations have 'AA', 'B%', or 'CA' kind of double letter transition pairs. Because of this update, the total state decreased by one. The accuracy increases significantly while having a trade-off on memory. In the single letter version, the memory complexity was $O((alphabet\ size)\ *\ (max\ word\ length))$, however, it is $O((alphabet\ size)^2\ *\ (max\ word\ length))$ in double letter version. Because of the double pair difference in this version, the probability calculation function changed as $freq\ of\ (x_i x_{i+1})\ /\ freq\ of\ (x_i\%)$ where x_i is a character. Additionally, the update operation of the words change as $freq\ of\ (x_i x_{i+1})\ +=\ 1$, $freq\ of\ (x_i\%)\ +=\ 1$, $freq\ of\ (\%\%)\ +=\ 1$, and $freq\ of\ (\%x_{i+1})\ +=\ 1$ for ith state where the ith letter of x is x_i and $i\ +\ 1$th letter of x is x_{i+1}. The '$' and '#' characters are also added at the beginning and end of the strings to make implementation and understanding easier. An example Double-Letter HMM can be seen in Figure 3 for the word set {"ahmet", "ali"}. As can be seen, the state count decreased by one if we do not count the two added transitions due to the '$' and '#' characters. The query pseudocode is nearly the same logically, again branching to other states is necessary if there is a '%' character. Otherwise, the states should be traversed linearly. Apart from these, this time instead of checking one letter in each state, we need to check two consecutive letters in each

state. The pseudocode in this approach is nearly the same as the single-letter approach and the branching strategy can vary from implementation to implementation.

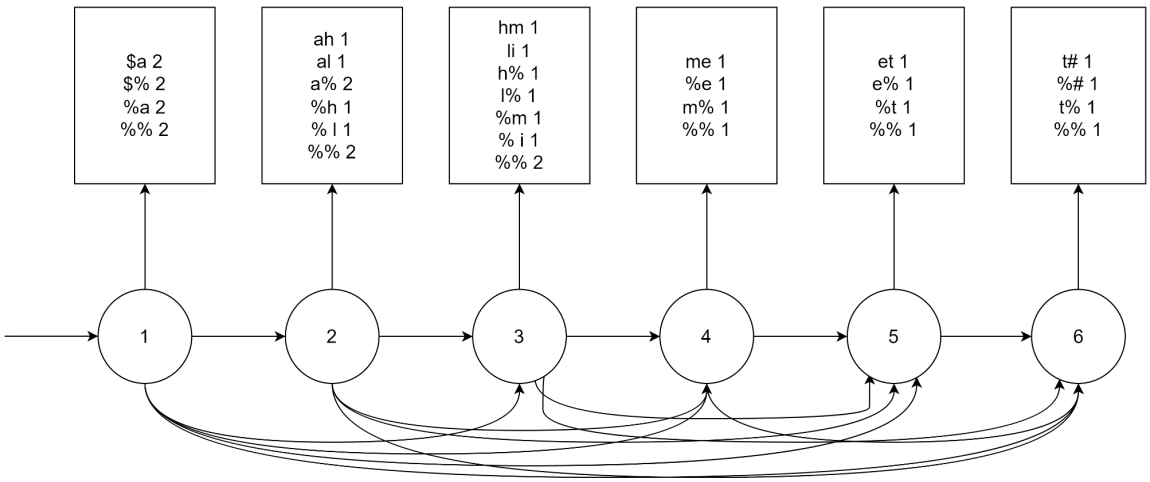

Figure 3: Example Extended Double-Letter HMM

It is obvious that this approach increases the accuracy significantly according to the Single Letter HMM approach. The details of these will be examined in the Experimentation part. However, due to the fact that the Double Letter HMM eliminates the unnecessary underestimation problem, it is more useful to use this approach instead of the single-letter one. Additionally, the accuracy might be increased by using Triple Letter HMMs, however, the memory consumptionA increases significantly in that case, which is $O((alphabet\ size)^3 * (max\ word\ length))$.

The pseudocode of the revised version for Extended Double-Letter HMM can be seen in the Algorithm 2. The edge cases are getting larger when the letter size of observations increased. For that reason, the if-else branches are getting much more when we increased it. In addition to all of these, a dynamic programming approach can be embedded to query function to bound the time complexity by $O(word\ size\ *\ state\ size)$. If the calculation already made for a specific $word\ index$ and $state\ index$ pair, it is unnecessary to make the calculation again and again. It is crucial to use dynamic programming if there is $\%s_1\%s_2\%s_3\%s_4\%s_5\%$ type of queries. Due to the branching fact, the time complexity can go up to exponentials, and it is better to use memoization in that sense.One of the most important drawbacks of this algorithm is it also enters the underestimation or overestimation mode when there are too many unique patterns overlapping with each other. An example of this overlapping scenario occurs in word-set {“gurkan”, “serkan”, “turhan”}, and imagine that we have a query “gurkan”. The calculation will be held as follows: $1/1 * 2/2 * 2/3 * 2/2 * 3/3 = 2/3$. In this scenario, an overestimation occurred instead of $1/3$, and the main reason is the overlap between the word set strings. As we mentioned before, this overestimation or underestimation error can be decreased by using the triple letter approach with a memory tradeoff.

In the following section, the comparison of these algorithms will be held by using different datasets and query sets for different scenarios.

```
function get_probability_helper(dp, word, branches, word_index, state_index):
  if word_index == word.size() - 1:
    return 1.0
  if dp contains {word_index, state_index}:
    return dp[{word_index, state_index}]
  if there are no observations for the specific state:
    return 0.0
  else:
    prob = states[state_index][word.substr(word_index, 2)] /
      (double)states[state_index][word.substr(word_index, 1) + '%']
    if the current letter does not have a preceding %:
      return dp[{word_index, state_index}] =
          prob * get_probability_helper(dp, word, branches, word_index + 1, state_index + 1)
    else:
      sub_prob = 0.0
      remaining_letters = word.size() - 1 - word_index
      sub_prob += (states[state_index][word.substr(word_index, 1) + word.substr(word_index+2, 1)]/
              (double)states[state_index][word.substr(word_index, 1) + '%']) *
              get_probability_helper(dp, word, branches, word_index + 2, state_index + 1);
      for each i from state_index + 1 to states.size() - remaining_letters:
        sub_prob += get_probability_helper(dp, word, branches, word_index + 1, i)
      res_prob = min(1.0, prob * sub_prob)
      return dp[{word_index, state_index}] = res_prob
```

Algorithm 2: Extended Double-Letter HMM Query Function

4. Experimental Results

In this section, we will present the experimental results of our work. The experiments are done using a personal laptop Huawei 14 which has an AMD Ryzen 7 CPU and 16 GB Ram. In this section, we compared our results with the state-of-art algorithms, Sequential-pattern based Histogram (SPH), and Lower Bound Estimation (LBS) that are proposed in the papers [9], and [8]. Since we do not have the code of these algorithms, we compared our results using the same dataset DBLP Author Dataset, however, some of the results might differ since they use a better setup.

We will only perform the experiments on Extended Double-Letter HMM because we already mentioned that using Single-Letter HMM algorithms will cause less accurate results.

4.1. Dataset

The DBLP Author dataset contains more than 4M author names. We eliminate some of them because they are shortened, and the remaining author names are around 3.5 million. We also create different datasets by sampling the values of the DBLP Author dataset. The datasets sizes that are used in experiments are 1k, 100k, 800k, and 3.5M in order. Also, the results of the 800k

dataset are compared with the results of the experiments that are held in [9]. The dataset has strings that have 13.73 lengths on average, 2 lengths at minimum, and 57 lengths at maximum.

4.2. Query Dataset

In the paper [9], they proposed three different query types which are:

- $\%w_1\%w_2\%$ type of queries where w_1 and w_2 are 5 and 12 character strings (named as 2-Group Queries).
- $\%w_1\%w_2\%...\%w_n$ type of queries where w_is are one or more character strings (named as More-Group Queries).
- The last one is negative queries, which means the result of the query set should be zero in real.

We created several query types also getting inspired from the paper to be compared. The first two query types are the same as the first two that are proposed in the paper, and the others are as follows:

- Instead of negative queries, we run our tests on exact queries, which means that the query string is in the dataset and there is only one of them (named as Exact Queries).
- $w\%$ type of prefix queries, where w is a character with a length more than 4 (named as Prefix Queries).
- $w\%$ type of prefix queries, where w is a character with a length between 1 and 4 (named as Short-Prefix Queries).
- $\%w$ type of suffix queries, where w is a character with a length more than 4 (named as Suffix Queries).
- $\%w$ type of suffix queries, where w is a character with a length between 1 and 4 (named as Short-Suffix Queries).

You can also see the namings in parentheses that will be used throughout this section.

4.3. Error Calculation

There are several ways of measuring how the proposed algorithms are accurate. We will use one of the most preferred calculation methods which is relative error calculation. The calculation formula of the relative error is $|Value_{real} - Value_{measured}| / Value_{real}$.

4.4. Accuracy Comparison by the Dataset Size

In this section, we compare our results with each other in different datasets that have different sizes. The aim is to find a correlation between dataset size and the accuracy of the queries.

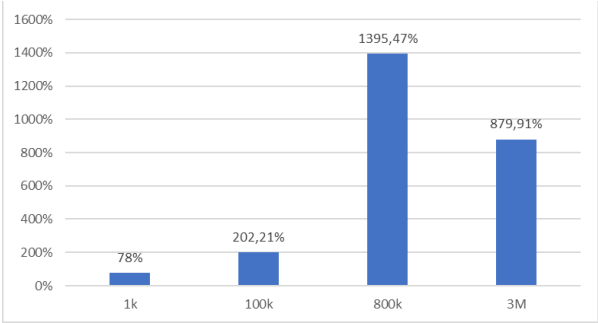

Figure 4: Relative Errors of the 2-Group Queries

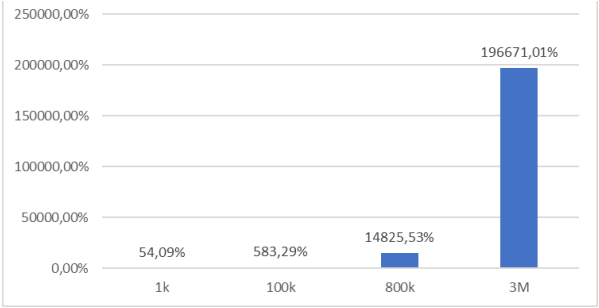

Figure 5: Relative Errors of the More-Group Queries

In Figure 4 and Figure 5, the relative errors of the $\%w_1\%w_2\%$ and $\%w_1\%w_2\%...\%w_n\%$ type of queries can be seen. According to these figures, we can observe that the accuracy is correlated with the dataset sizes. Also, it is not meaningful to use this algorithm for More-Group Queries if the dataset is larger than 800k because the relative errors are significantly high. Especially, if we have a small amount of dataset size, this algorithm can be useful because the relative errors are getting lower when the dataset size decreased.

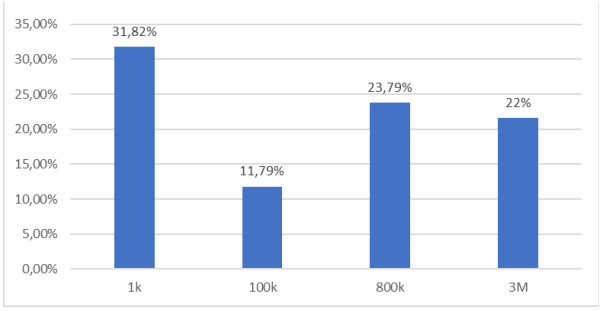

Figure 6: Relative Errors of the Exact Queries

In Figure 6, the relative errors are close to 50%, and it seems that the HMM-based Algorithms work best among the compared queries.

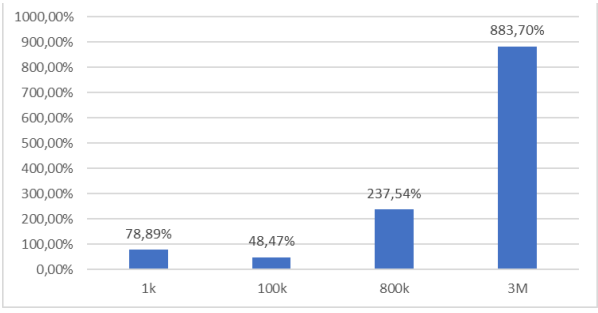

Figure 7: Relative Errors of the Prefix Queries

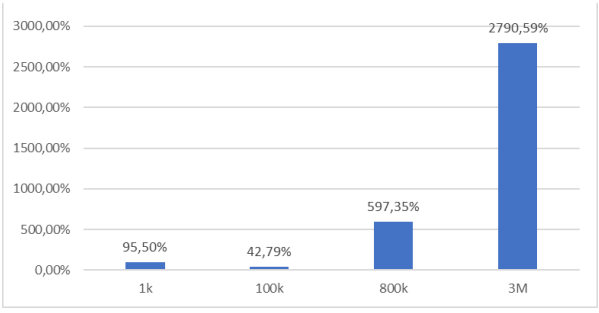

Figure 8: Relative Errors of the Short-Prefix Queries

In Figure 7 and Figure 8, the relative errors of the Prefix Queries can be observed. According to the tables, it can be said that if the word length of the prefix increases, then the results get more accurate. Also, the relative error increases, most of the time, when the dataset size increases. If the dataset size is smaller than 100k, the algorithm is significantly useful when we consider that other algorithms SPH and LBS can not make an estimation for Prefix Queries.

In Figure 9 and Figure 10, the relative errors of the Suffix Queries can be analyzed. Besides that algorithm works efficiently for the Prefix Queries, it does not work for Suffix Queries with a similar accuracy. The relative errors increase significantly when the dataset gets larger than 800k. To fix the accuracy issue, another approach might be to build one more HMM network with reverses of strings. The query for the reverse HMM will have similar accuracies with the Prefix Queries in that case. The memory complexity will not be affected too much since it already has low memory consumption.

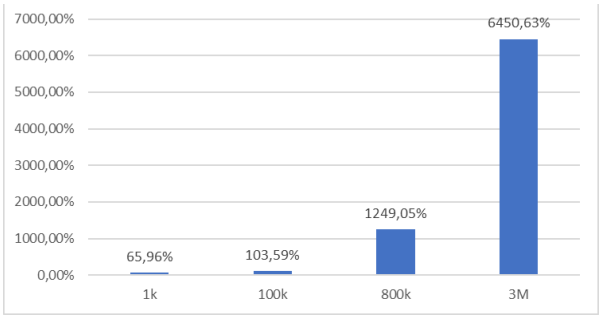

Figure 9: Relative Errors of the Suffix Queries

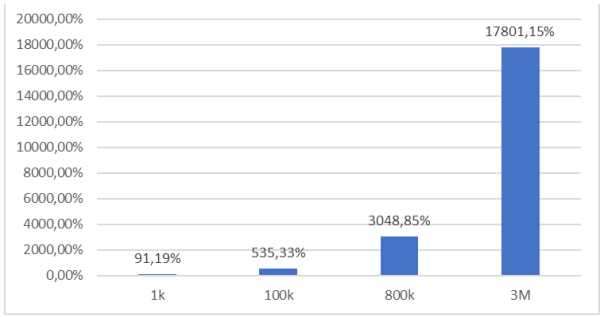

Figure 10: Relative Errors of the Short-Suffix Queries

To conclude this section, the proposed algorithm is proper for most of the query types when the unique dataset size is smaller than or equal to 100k. When it exceeds more, the results get remarkably less accurate.

4.5. Runtime and Memory Consumption Comparison by Dataset Size

In this section, we will analyze the build time, query optimization time, and memory consumption of the algorithm. Note that the implementation of the algorithm is done using CPP, which means that there is no overhead due to the programming language and probably the one that consumes minimum memory.

In Figure 11 and Figure 12, the total time in the build phase and memory consumption during program work are represented in order. The total time increases proportionally with the dataset size because it needs to process all the words one by one. Besides, memory consumption increases proportionally, again, with the dataset size, however, the increase rate is linear with the dataset size. There is a small amount of change while the dataset size increases aggressively.

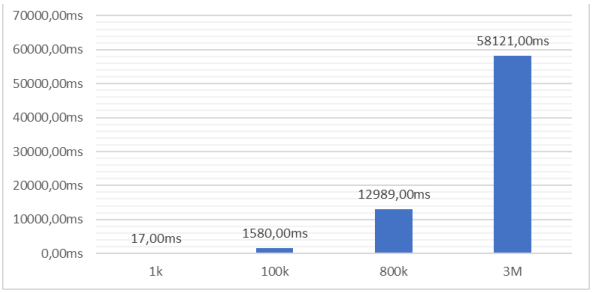

Figure 11: Total Time in Build-Phase

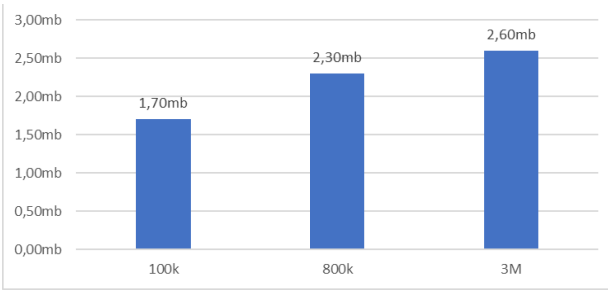

Figure 12: Memory Consumption on Build and Query Phases (in Total)

The query optimization times are also represented in Figure 13 and Figure 14. The algorithm works pretty effectively for each query because the algorithm complexity is only dependent on alphabet size and max word length which are actually constants. Even in the worst case, it works effectively. There is a slight increase for the More-Group Queries when compared to the 2-Group Queries. The main cause is branching occurs more for the More-Group Queries and it can be seen as worst-case behavior.

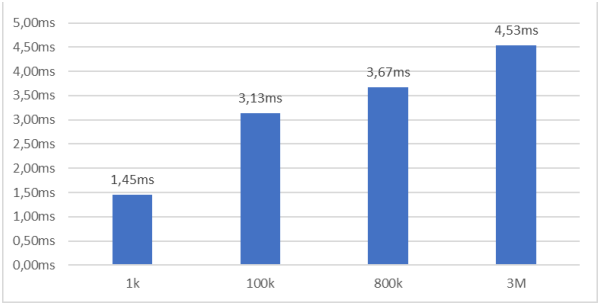

Figure 13: Query Optimization Time of 2-Group Queries

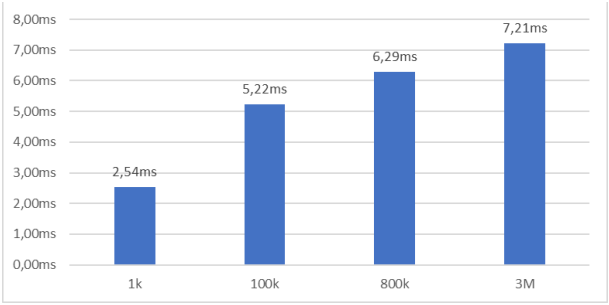

Figure 14: Query Optimization Time of More-Group Queries

4.6. Comparison with LBS and SPH

In this section, we will compare the result of our algorithms with the state-of-art algorithms which are LBS and SPH. At first, the accuracy comparison will be analyzed, and then, time and memory usage comparisons will be held.

In Figure 15 and Figure 16, the relative errors are represented in the tables for the queries in format $\%w_1\%w_2\%$ and $\%w_1\%w_2\%...\%w_n$ in order. When we look at the tables, it is obvious that the result of our algorithm does not accurate when compared to LBS and SPH. They have a significant advance. The SPH algorithm performs 18 times better than HMM approach for 2-Group Queries, and 203 times better for More-Group Queries. The most likely reason is our

branching strategy does not work as well as in theory. Because we also observed that the error rate increases significantly when the dataset gets larger. The branching strategy might be changed to fix the accuracy.

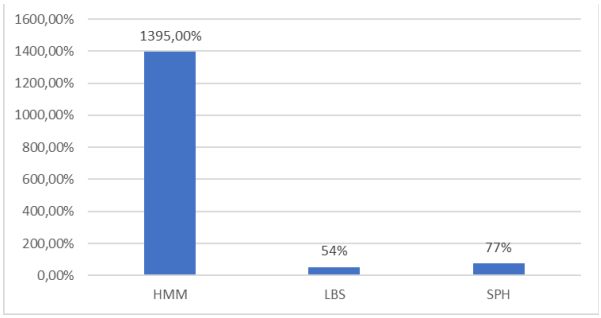

Figure 15: Relative Errors of 2-Group Queries

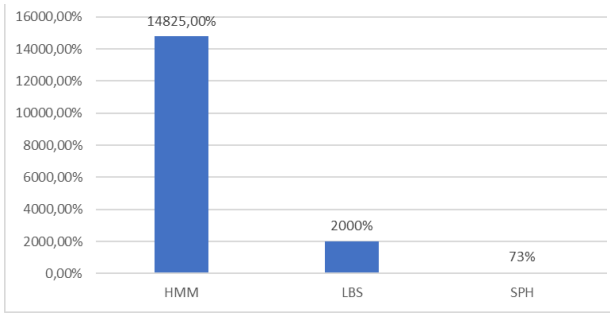

Figure 16: Relative Errors of More-Group Queries

Besides, it is important that the LBS and SPH algorithms do not work for Exact or Prefix types of queries. Our algorithms work best when the word lengths do not vary so much, and if the missing part of the queries contains a small amount of characters. For example, if we only have words that have a word length between 30 and 40, and the % signs cover 2-3 characters instead of covering all of the word, i.e. %s% where s is only one character.

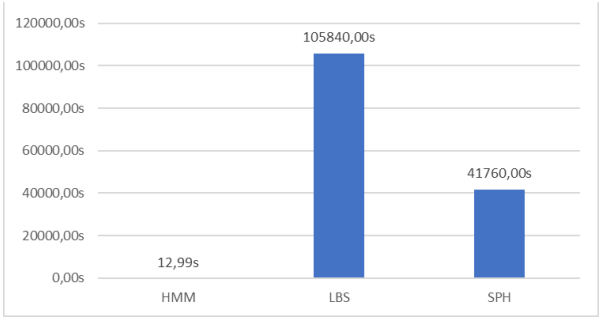

Figure 17: Build Time Comparison of the Algorithms

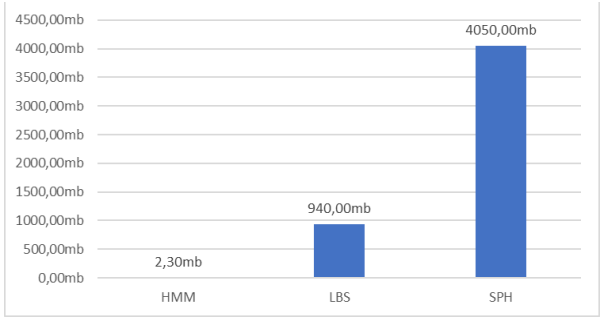

Figure 18: Memory Consumption Comparison of the Algorithms

In Figure 17 and Figure 18, the build time and memory consumption during build phases can be seen. The HMM approach uses a significantly smaller amount of resources during the build phase, and the query optimization time is only 12.99 seconds. HMM algorithm runs 3212 times quicker than the SPH algorithm and uses 1760 times fewer memory resources in the build phase. The algorithms have significant advances in these, and we can extend our algorithm as we wish to develop a more accurate algorithm.

Moreover, the query optimization time works 2 times faster than the SPH algorithm as is seen in Figure 19.

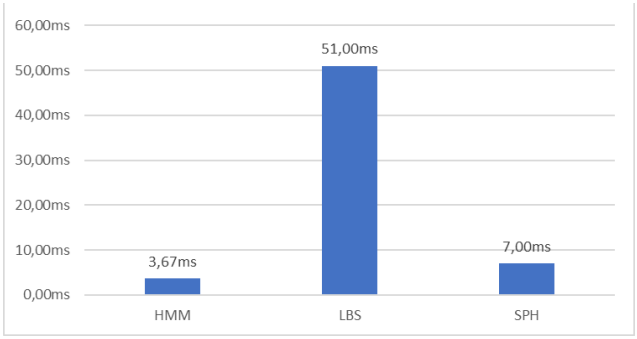

Figure 19: Query Optimization Time Comparison of the Algorithms

To sum up, the HMM algorithm does not results as accurately as SPH and LBS algorithms, however, there is room for improvement when we consider how it uses significantly fewer amount of resources and works faster. The HMM approach will lead to new algorithms in this area.

5. Conclusion

In this paper, we proposed a novel approach for predicting string selectivity in SQL LIKE queries by leveraging Hidden Markov Models (HMMs) and Markov Chains. We introduced two versions of our algorithm: Single-Letter HMM and Extended Double-Letter HMM. The Single-Letter HMM version mapped letters with their indexes and used state transitions and observation values to estimate selectivities. However, this version had limitations in handling wildcard characters and underestimated exact queries. To overcome these, we developed the Extended Double-Letter HMM version, which introduced double-letter transition pairs and improved accuracy significantly, while having a trade-off for memory complexity.

Through experiments conducted on the DBLP Author dataset, we compared our results with state-of-the-art algorithms, namely Sequential-pattern based Histogram (SPH) and Lower Bound Estimation (LBS). Although we did not have access to the exact implementation of these algorithms, our comparisons provided valuable insights into the performance of our approach.

The experimental results demonstrated the effectiveness of our Extended Double-Letter HMM algorithm in predicting string selectivity. It underperformed the other algorithms in terms of accuracy and while outperforming in terms of efficiency. We observed that there are still areas for improvement, specifically for the 2-Group and More-Group queries. To address the issues, future research could be needed.

In conclusion, our research contributes to the field of string selectivity prediction in database systems by introducing an innovative approach using HMMs and Markov Chains. This paper will lead to new approaches for further investigation into optimizing query selectivity estimation, and we hope it inspires future research in this domain.

References

[1] Knuth, Donald E., et al. "Fast Pattern Matching in Strings." *SIAM Journal on Computing*, vol. 6, no. 2, 1977, pp. 323–350., https://doi.org/10.1137/0206024.

[2] Jurafsky, Dan, and James H. Martin. "Hidden Markov Models." *Speech and Language Processing*, Pearson Prentice Hall, Harlow, 2014.

[3] Mahfuz, Fariha. *MARKOV CHAINS AND THEIR APPLICATIONS*, *University of Texas at Tyler*, 2021, https://scholarworks.uttyler.edu/math_grad/10/. Accessed 2023.

[4] Krishnan, P., et al. "Estimating Alphanumeric Selectivity in the Presence of Wildcards." ACM SIGMOD Record, vol. 25, no. 2, 1996, pp. 282–293., https://doi.org/10.1145/235968.233341.

[5] Jagadish, H. V., et al. "Substring Selectivity Estimation." Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, 1999, https://doi.org/10.1145/303976.304001.

[6] Chaudhuri, S., et al. "Selectivity Estimation for String Predicates: Overcoming the Underestimation Problem." Proceedings. 20th International Conference on Data Engineering, 2004, https://doi.org/10.1109/icde.2004.1319999.

[7] Li, Dong, et al. "Selectivity Estimation for String Predicates Based on Modified Pruned Count‐Suffix Tree." Chinese Journal of Electronics, vol. 24, no. 1, 2015, pp. 76–82., https://doi.org/10.1049/cje.2015.01.013.

[8] Lee, Hongrae, et al. "Approximate Substring Selectivity Estimation." Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, 2009, https://doi.org/10.1145/1516360.1516455.

[9] Aytimur, Mehmet, and Ali Çakmak. "Estimating the Selectivity of like Queries Using Pattern-Based Histograms." TURKISH JOURNAL OF ELECTRICAL ENGINEERING & COMPUTER SCIENCES, vol. 26, no. 6, 2018, pp. 3320–3335., https://doi.org/10.3906/elk-1806-96.