Estimating the selectivity of LIKE queries using pattern-based histograms

MEHMET AYTİMUR

ALİ ÇAKMAK

Research Article

Estimating the selectivity of LIKE queries using pattern-based histograms

Mehmet AYTİMUR[1]*, **Ali ÇAKMAK**[1]

[1]Department of Computer Science, Faculty of Engineering, İstanbul Şehir University, İstanbul, Turkey

Abstract: Accurate cost and time estimation of a query is one of the major success indicators for database management systems. SQL allows the expression of flexible queries on text-formatted data. The LIKE operator is used to search for a specified pattern (e.g., LIKE "luck%") in a string database. It is vital to estimate the selectivity of such flexible predicates for the query optimizer to choose an efficient execution plan. In this paper, we study the problem of estimating the selectivity of a LIKE query predicate over a bag of strings. We propose a new type of pattern-based histogram structure to summarize the data distribution in a particular column. More specifically, we first mine sequential patterns over a given string database and then construct a special histogram out of the mined patterns. During query optimization time, pattern-based histograms are exploited to estimate the selectivity of a LIKE predicate. The experimental results on a real dataset from DBLP show that the proposed technique outperforms the state of the art for generic LIKE queries like $\%s_1\%s_2\%...\%s_n\%$ where s_i represents one or more characters. What is more, the proposed histogram structure requires more than two orders of magnitude smaller memory space, and the estimation time is almost an order of magnitude less in comparison to the state of the art.

Key words: Selectivity estimation, histograms, data management, sequence mining

1. Introduction

One of the key reasons for the success of relational database management systems is their advanced query optimization capabilities. The query optimizer explores all or a subset of possible execution plan alternatives and determines the most efficient way to execute a given query. With the explosion of the Internet and text-based data, the role of the query optimizer is even more critical to efficiently query the huge amounts of textual data. A commonly observed property of such datasets is that they are often unclean and contain misspellings, typographical errors, etc. Hence, rather than exact equality string predicates, often, flexible patterns are preferred to search in such textual data piles. SQL provides the LIKE operator to enable approximate string searches. As an example, consider a table in a database that stores customer records such as name, age, salary, etc. Suppose a user wants to retrieve all information about customers whose first names start with the substring "Lucia". In SQL, such a query is expressed as follows: *SELECT * FROM CUSTOMERS WHERE name LIKE 'Lucia%'.*

In query optimization, accurate and efficient predicate selectivity estimation with low cost (i.e. time and memory consumption) is instrumental to produce an optimized query execution plan. In this paper, we study the problem of estimating the selectivity of queries with LIKE predicates. In order to estimate the selectivity of wildcard predicates, various techniques (e.g., [1–4]) have been proposed in the literature. These techniques

---

*Correspondence: alicakmak@sehir.edu.tr

build summary structures to estimate the frequencies of string predicates. They usually process large sets of rows ids for each input query. Hence, these techniques may not perform well for generic LIKE queries that have the form $\%s_1\%s_2\%...\%s_n\%$ where s_i represents one or more characters.

In this paper, we propose a new type of pattern-based histogram structure. More specifically, we first mine sequential patterns from the corresponding text column offline before query time. Then we build a histogram structure out of the discovered sequential patterns and store it in the database catalog as part of column statistics. During query time, we search for a given LIKE predicate pattern in a query in the precomputed histogram, and accordingly, we come up with a selectivity estimation. The algorithm that we propose is called SPH (i.e. Sequential Pattern-based Histogram). Figure 1 summarizes the SPH approach. Part I demonstrates the steps that are performed before query time, possibly during statistics gathering time, and Part II shows the steps that take place online during query time.

We comprehensively evaluate our method on a real dataset from DBLP. Our results show that SPH dramatically outperforms the state of the art approach (i.e. LBS [2]) for queries with generic LIKE patterns in terms of estimation accuracy. Moreover, SPH requires two orders of magnitude less space both in memory and in the database catalog, which may be critical given that there may be millions of columns in a typical production database of a business. Besides, the selectivity estimation time of SPH is almost an order of magnitude less in comparison to the state of the art.

Our primary contributions in this part of our work is as follows:

- We propose a new frequent sequence pattern-based histogram structure that requires significantly less space than the index structure used in the state-of-the-art approach [2].

- Using the above histogram structure, we develop a LIKE query selectivity estimation technique that is over an order of magnitude more accurate than the state of the art for generic LIKE queries. Besides, for more specific LIKE queries, the accuracies are comparable. Meanwhile, the proposed technique requires dramatically less time and space during query optimization time.

- We generate different groups of test queries, and on a real dataset we comprehensively evaluate the accuracy, running time, and space overhead of the proposed technique in comparison to the state of the art.

The experimental results on a real dataset from DBLP show that:

- SPH outperforms the state of the art for generic LIKE queries like $\%s_1\%s_2\%...\%s_n\%$, where s_i represents one or more characters, by a significant margin (i.e. around 1200% on a mixed workload).

- The proposed histogram structure requires more than two orders of magnitude smaller memory space.

- The selectivity estimation time is almost an order of magnitude less in comparison to the state of the art.

The rest of this paper is organized as follows. In Section 2, we discuss the related work. Section 3 describes the construction of pattern-based histograms. In Section 4, we discuss the selectivity estimation algorithm using the proposed histograms. Section 5 presents the experimental results on a real dataset. Section 6 proposes some directions for future work, and we conclude in Section 7.
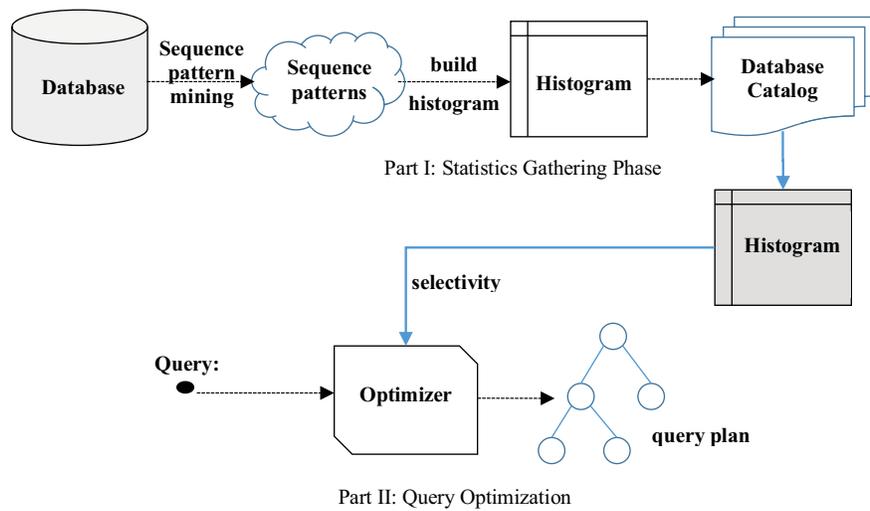
Figure 1. A high-level overview of SPH.

2. Related work

In query optimization, the estimation of selectivities for query predicates is required to predict the cost of possible alternative execution plans for a given query. Thus, the selectivity estimation of LIKE predicates in SQL queries has been studied extensively in the literature [1, 3, 5]. Suffix trees have been often used as a data structure for approximate string matching. For efficient storing and searching suffix trees on disk, Ferragina and Grossi [6] proposed the String B-tree. Krishnan et al. [1] employed pruned suffix trees to structure the text-based data in databases. Their proposed algorithm, KVI, assumes independence of substrings. To alleviate the problems due to this assumption, Jagadish et al. [4] proposed the MO algorithm, which is based on the Markov assumption and the concept of maximally overlapping substrings. Chaudhuri et al. [5] observed that MO often underestimates, and they introduced the CRT algorithm based on the Short Identifying Substring (SIS) assumption.

Histogram structure has been widely used to estimate the selectivity of different data types. To this end, several types of histograms are proposed in the literature. To et al. [7] used entropy-based histograms to estimate the selectivity of queries. They offered three algorithms (ME, MSE and, MB), which are based on information entropy, to build the histograms. Poosala et al. [8] offered equi-width and equi-height histogram structures based on different partitioning rules. Jagadish et al. [9] proposed optimal histograms. They developed algorithms that compute optimal bucket boundaries for the histograms. They showed that optimal histograms provide lower estimation error than regular histograms. In another study [10], wavelet-based histograms were exploited for selectivity estimation. These types of histograms provide more accurate results in comparison to the equi-width and equi-height histograms. Muralikrishna and DeWitt [11] proposed an algorithm to build multidimensional histograms to estimate the selectivity of multidimensional queries. The offered algorithm decreases the cost of the histogram building phase. Jin and Li [12] proposed SEPIA, which estimates the selectivity of string predicates based on edit distance. The edit distance between given two strings is the minimum number of edit operations (insert, delete, and substitute) to transform one string into the other one. The essence of their approach is to group strings in a database column into clusters and construct a separate histogram structure for each cluster. Then the selectivity of a LIKE query is estimated by using edit distance. Likewise, edit distance is used as a similarity function in [13–16] in the context of approximate string matching and set similarity joins.

Lee et al. [2] also used edit distance to find the similarity between a given query and a bag of strings. They proposed two algorithms, MOF and LBS, which are based on information stored in an N-gram table. MOF extends from the authors' similar earlier work [17] and employs q-grams. More specifically, it finds all minimal base-substrings of an input string and stores them in the N-gram table. Then MO is used to compute the selectivity of the string in a LIKE predicate. LBS is an extended version of MOF. It assigns a signature to each minimal base-substring and uses the N-gram table in combination with MO to estimate the selectivity of each minimal base-substring. Then, by applying set union and intersection operations on signatures, it computes the selectivity of a predicate string. Kim et al. [18] used a similar technique to estimate the selectivity of a string query. That is, they used inverted-gram indices to estimate the selectivity of q-grams. Similar to LBS, they assign a signature to the substrings, which are created through random permutations of row ids. Since this technique is very similar to LBS, and they both have almost the same results, we consider the LBS algorithm as the representative state-of-the-art approach and compare our approach to LBS experimentally in Section 5.

Furthermore, approximate string matching is also popular in spatial database research. As an example, Yao et al. [19] used MHRtree to efficiently answer approximate string matching queries in large spatial databases. Their technique is based on the min-wise signature and the linear hashing technique, which is also used in LBS, as well.

3. Pattern-based string histogram construction

In this section, we discuss our histogram construction process. We first present the sequential pattern mining step. It is followed by the compilation of a histogram out of the mined patterns.

3.1. Mining sequential patterns

Sequence pattern mining was first proposed by Agrawal and Srikant [20]. With its introduction, it became popular in data mining, and it has a broad area of applications such as discovering patterns in DNA to reveal protein-coding regions, stock market analysis, and web log click stream analysis. Sequence pattern mining is concerned with finding statistically relevant patterns in a dataset where the values are organized in a sequence. The problem of sequence pattern mining is defined as follows:

Def'n (Sequence Pattern Mining): *Find all sequences whose frequencies are greater than or equal to a given threshold, min-sup, in a sequence database.*

Table 1. An example sequence database.

Sequence id	Sequence
1	ACCBADE
2	BBECABAB
3	CAABCBEA
4	DABAACBCE

Example. *Consider the database shown in Table 1, which has four sequences. Assume that the min-sup threshold is 3. Then the complete set of the frequent sequences and their corresponding frequencies are as follows: {AAB:3, CB:4, CBA:3, ABB:3, BCB:3, BE:4, AAE:3, BBE:3, CCE:3, ACBE:3, ABA:4, A:4, AA:4, AB:4, AC:3, ACB:3, B:4, BA:4, BB:3, BC:3, C:4, CA:4, CC:3}.*

For large datasets, mining all possible patterns is not considered effective due to the enormous number of possible patterns. Instead, mining a special subset of patterns whose frequency is greater than all of its super-patterns is often preferred over mining the full set of patterns. Such patterns are called closed patterns. Several approaches [21–24] have been proposed to mine only closed patterns and leave the others. As part of our proposed method, any existing sequence pattern mining technique may be employed. In this paper, we choose to use the BIDE algorithm [25], as it manages to avoid a candidate maintenance step during pattern mining. Hence, it spends less memory and runs faster than the other competitors.

Def'n (Frequent Closed Sequence): *Assume that S_a and S_b are two sequences. If sequence S_b contains sequence S_a , then S_b is a supersequence of S_a . If a sequence is frequent and has no supersequence with the same frequency, then it is called a frequent closed sequence.*

BIDE first scans the database once to find all frequent sequences of length 1 (i.e. a letter), and then it builds a pseudo-projected database for each frequent letter. It uses each frequent letter as a prefix and employs a back-scan pruning method to speed up the mining. Furthermore, it uses a backward-extension pruning method to narrow down the search space. Then it applies the same procedure for the newly discovered frequent sequences that have length greater than 1.

Example. *Consider the database shown in Table 1 with four rows. Assume that the minimum support threshold is 3. The complete set of the frequent closed sequences is {AAB:3, CB:4, CBA:3, ABB:3, BCB:3, BE:4, AAE:3, BBE:3, CCE:3, ACBE:3, ABA:4}. Note that there are 11 unique frequent closed sequences. In comparison, regular sequence mining produces 23 patterns as shown in the previous example.*

3.2. Histogram construction

Histograms are widely used to represent the data distribution in a database column. Database systems use histograms to summarize skewed data stored in a column, and subsequently provide result size estimates for queries.

Frequency, height-balanced, and hybrid types of histograms are widely used. In a frequency histogram, each distinct column value corresponds to a single bucket of a histogram. In a height-balanced histogram, each bucket contains the same number of rows. A hybrid histogram combines characteristics of both height-based and frequency histogram. That is, it distributes values so that no value occupies more than one bucket, and it stores the frequency value for each endpoint in the histogram. Usually, the number of distinct values (NDV) and the distribution of data determine the type of the histogram to be used. If the NDV in a column is at most the number of available buckets (usually a user-determined parameter), then frequency histograms are preferred. Otherwise, height-balanced or hybrid histograms are preferred.

From this point on, for brevity, we discuss the construction and use of only hybrid histograms. The discussed methods directly apply to the other types of histograms as well with no or minimal modification. In order to build a histogram out of the mined sequence patterns, we first sort the patterns. Then the capacity of each bucket is determined. That is, the cumulative total frequency of patterns is considered as the total number of rows. In order to compute bucket capacity, the total number of rows is divided by the number of buckets that the histogram is allowed to have. Let the bucket capacity be C. In the next step, bucket endpoints are determined. To this end, the sorted pattern list is traversed starting from the first one while keeping a running sum S of pattern frequencies. Whenever S ≥ n * C where n is an integer with initial value n = 1, we set that particular pattern as the bucket endpoint and update n's value as n = floor(S/C) + 1 (floor of a real number n returns the largest integer that is smaller than n). Then we continue the same process until the number of the

determined histogram endpoints becomes equal to the total number of buckets allowed for the histogram or all the patterns are consumed. We give an example.

Example. *Consider the set of closed sequence patterns that are mined with minimum support threshold 3 from the string database in Table 1: {AAB:3, CB:4, CBA:3, ABB:3, BCB:3, BE:4, AAE:3, BBE:3, CCE:3, ACBE:3, ABA:4}. Assume that the specified number of buckets is 4. Figure 2 shows the steps of histogram construction out of this pattern set. In the first step, patterns are sorted in alphabetical order. Total frequency of patterns is 36. In step 2, the bucket capacity is determined as 36/4 buckets = 9. In step 3, the bucket endpoints are located as described above. Step 4 shows the resulting histogram. The first column in this table lists the cumulative frequency of all patterns in the sorted pattern list of step 1 up until the bucket endpoint value (i.e. a pattern). Column 2 stores the bucket endpoint values, and column 3 keeps the corresponding individual frequency for each bucket endpoint value in column 2.*
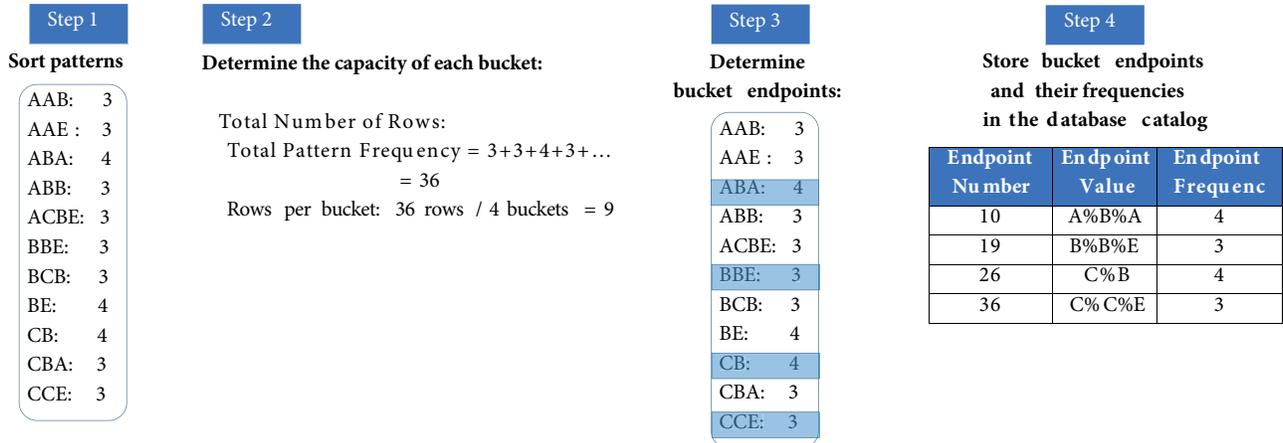
Step 1	Step 2	Step 3	Step 4
Sort patterns	**Determine the capacity of each bucket:**	**Determine bucket endpoints:**	**Store bucket endpoints and their frequencies in the database catalog**

Step 1 — Sort patterns:

AAB:	3
AAE :	3
ABA:	4
ABB:	3
ACBE:	3
BBE:	3
BCB:	3
BE:	4
CB:	4
CBA:	3
CCE:	3

Step 2 — Determine the capacity of each bucket:

Total Number of Rows:
Total Pattern Frequency = 3+3+4+3+…
= 36
Rows per bucket: 36 rows / 4 buckets = 9

Step 3 — Determine bucket endpoints:

AAB:	3
AAE :	3
ABA:	4
ABB:	3
ACBE:	3
BBE:	3
BCB:	3
BE:	4
CB:	4
CBA:	3
CCE:	3

Step 4 — Store bucket endpoints and their frequencies in the database catalog:

Endpoint Number	Endpoint Value	Endpoint Frequenc
10	A%B%A	4
19	B%B%E	3
26	C%B	4
36	C%C%E	3

Figure 2. Pattern-based histogram construction with bucket count 4.

Note that, in the histogram, bucket endpoint values have a '%' symbol added between each letter of the corresponding pattern. This symbol represents zero or more characters, as in SQL LIKE syntax. This transformation is consistent with the semantics of a sequence pattern, since the consecutive letters in a pattern may have other characters between them in each of the pattern's occurrences in a database. Once a pattern-based histogram is computed offline during database statistics gathering, it is stored in the database dictionary/catalog to be later used during query optimization time for selectivity estimation.

4. Selectivity estimation

In this section, we discuss the details of estimating LIKE query selectivities using the pattern-based histograms. Given a query with a LIKE predicate p, our algorithm visits all histogram buckets and check if there is any bucket endpoint that 'matches' p. Here, there may be two different possible ways of matching between a predicate p and bucket endpoint b: (i) p exactly matches b, or (ii) p is encapsulated in b. We next formally define these matching types.

Def'n (Exact Match): *If a LIKE predicate pattern p is exactly the same as the bucket endpoint b, then p is considered to exactly match b.*

Example. *Consider a LIKE query predicate pattern 'A%B%C'. In order to have an exact match in a histogram, the histogram must include 'A%B%C' as an endpoint value for a bucket.*

Def'n (Encapsulated Match): *Let a LIKE predicate pattern p have the form $s_1 s_2 .... s_n$ where each s_i is either a regular character from the corresponding alphabet that forms the strings in the database or is a wildcard character (i.e. % or _). For any pair of s_i, s_j from p where $i < j$, if both s_i and s_j are included in a bucket endpoint b, and s_i comes before s_j in b, then there is encapsulated match between p and b.*

Example. *Consider a LIKE query predicate pattern p = 'A%B%C'. Then there is an encapsulated match between p and any of the following histogram bucket endpoints: '$S_i A\%B\%CS_j$ %' or '$S_i A\%S_j B\%CS_k$', where S_i, S_j, and S_k represent one or more characters.*

According to the type of the match between a LIKE predicate pattern p and histogram bucket endpoints, we estimate the selectivity as follows:

- *Exact match*: If there is a histogram bucket endpoint b that matches p exactly, then the endpoint frequency of b is set as the estimated number of rows that would match p in the database.

- *Encapsulated match*: If there is encapsulated match between p and a set B of histogram bucket endpoints, then the average endpoint frequency of bucket endpoints in B is set as the estimated number of rows that would match p in the database.

- *No match*: If there is no exact or encapsulated match between p and histogram bucket endpoints, then the estimated selectivity of p is set to t% of the minimum support threshold where t is experimentally determined based on the data. We experimented with 1%, 5%, 10%, 15%, and 20%, where 10% provides the best accuracy. Hence, we consider t = 10 in this work.

Algorithm 1 shows the pseudocode for the above summarized approach. The first three lines represent the histogram building phase. It first mines all frequent closed sequences in a given database and then builds a histogram out of the mined patterns. The for loop from lines 4 to 10 checks the above three cases and sets the estimated frequency for a LIKE query predicate pattern p. Finally, it returns the estimated frequency in line 11.

Algorithm 1 SPH

Require: A database D; An SQL LIKE predicate pattern p; Minimum support threshold *minsup*;
Ensure: The estimated frequency for p;
1: FCS=BIDE(D, minsup);
2: freq=0
3: hist=Histogram(FCS);
4: **for** each bucket b $\in$ hist **do**
5:     **if** p exactly matches b **then**
6:         freq=endpoint_freq(b);
7:     **else if** p encapsulated|matches a set B of bucket endpoints **then**
8:         freq $=(\sum$ endpoint_freq$(b_i))$ / $|B|$ where $b_i \in$ B
9:     **else**
10:         freq= 0.1*minsup
11: **return** freq;

Example. *Consider the database D in Table 2 that contains 9 rows. With minimum support threshold 3, 1015 frequent closed patterns are mined from D. Let the specified bucket number be 10. Table 3 shows the resulting pattern-based histogram constructed out of the mined sequence patterns.*

Table 2. An example sequence database D.

Sequence id	Sequence
1	ACDFECBAAECAEEABB
2	BBECAEBABBBBDCAE
3	CAADCBEACAADBE
4	DABAACBCECABBAE
5	AECCFBADEFDEDA
6	BABECABABABCBCE
7	CAABCFBEADEBAEBDE
8	DAFBAACABCEABD
9	ACECBAADEAEAEC

Table 3. Constructed pattern-based histogram for the database in Table 2.

medskip

Endpoint number	Endpoint value	Endpoint frequency
385	A%A%E%A	8
765	A%B%A%E%A%B	4
1147	A%C%B%A%B%B	4
1527	A%D%B%E%E	3
1909	B%A%A%A%C	4
2291	B%B%E%A%B%D	4
2676	B%E%C%A%E	6
3058	C%A%C%E	7
3440	C%B%E%C%E	4
3804	D%E%B%A%E	3

Exact match case: Assume that LIKE query predicate p is 'C%A%C%E'. p exactly matches the 8th bucket of the histogram. Hence, the estimated selectivity is 0.7 (i.e. 7/10, where 7 is the corresponding bucket endpoint frequency and 10 is the number of rows in the database).

Encapsulated match case: Assume that p is 'B%A%B'. There is an encapsulated match between p and the 2nd, 3rd, and 6th bucket endpoints of the histogram. Hence, the average selectivity is 0.4 (i.e. (4+4+4)/3 = 4 is the average frequency of matching endpoints and it is then divided by the total number of rows in the database: 4/10 = 0.4).

No match case: Assume that p is 'D%A%D%E'. Since p does not match any bucket endpoint in the histogram, the estimated frequency is set as 10% of the minimum support threshold (i.e. 0.3). Hence, the estimated selectivity of the query is 0.03 (i.e. 0.3/10).

5. Experimental evaluation

In this section, we present an empirical evaluation of the proposed approach. All the tests are done on a Dell R720 machine with 2x XEON E-5-2620v2 2.10 GHz CPU and 80 GB RAM.

5.1. Dataset

We perform various experiments using the DBLP author dataset. The DBLP author dataset contains 800,000 rows with full author names. The average, minimum, and maximum lengths are 22.5, 18, and 60, respectively.

5.2. Test query set

In order to test the accuracy of selectivity estimation, we generate four distinct sets of queries with different characteristics. There are 100 queries in each group, except that the fourth group contains 24 queries.

- The first group of test queries is constructed by following the same approach as described in the LBS paper [2]. That is, it contains LIKE queries in the form of %w% and $\%w_1\%w_2\%$ where w_i represents a word with length between 5 and 12. To construct this query set, we randomly select one or two words of length between 5 and 12 from the database and introduce 0 to 2 underscore characters (i.e. '_') in each word at random positions, where '_' represents a wildcard that matches any single character. The average, minimum, and maximum lengths for the query predicates in this group are 6.71, 5, and 17. The average selectivity for the first group of queries is 4.77%.

- In the second group, we aim to generate more generic LIKE queries in the form of $\%s_1\%s_2\%.....\%s_n\%$ where s_i represents one or more characters. More specifically, we randomly get a row from the database, and we draw a random number, k, between 3 and the length of the row. Then we randomly remove k characters from the row and insert 2 to 8 '%' symbols between the remaining characters. The average, minimum, and maximum lengths for the query predicates in this group are 8.4, 3, and 16. The average selectivity for the second group of queries is 2.67%. We give an example.

  Example: Assume that the randomly chosen row is 'LUCKRICIA', the number of characters to be removed is 4, and the number of % symbols to be inserted is 3. We first randomly remove 4 characters from the string, and then the remaining characters are 'LKRIC'. Then we randomly insert 3 '%' symbols between the remaining characters, and the final LIKE query is 'L%KR%I%C'.

- As for the set of negative queries that do not return any matching rows, we generate queries in a manner similar to the second group of queries, but with a low number of '%' symbols. More specifically, instead of 2 to 8, we randomly insert 1 to 3 '%' symbols. Out of 100 generated queries, 24 of them are truly negative queries with 0 matching rows. Hence, this set, in its final form, contains 24 queries.

5.3. Evaluation metrics

In order to evaluate the accuracy of selectivity estimation, we employ two metrics. The first one is relative error, which is defined as f_{true} $\|f_{\text{true}} - f_{\text{est}}\|/f_{\text{true}}$, where f_{true} is the actual number of rows that should be returned if the query is executed, and f_{est} is the estimated cardinality. The second one is absolute error, which is defined as $|f_{\text{est}} - f_{\text{true}}|$. In order to prevent the accuracy from being distorted by queries that have small actual value, we exclude those queries with actual frequencies smaller than 10 following the same approach as in [2]. For such queries and negative queries, to estimate the accuracy, we employ the absolute error metric.

This section presents our experimental results. In the first part, we evaluate different aspects of our approach. In the second part, we compare our approach to the state of the art in terms of estimation accuracy, query time, and space overhead.

5.3.1. The effect of the minimum support threshold

Minimum support threshold, during frequent sequence mining, directly affects the number of patterns in the result set. When the minimum support threshold is too low, there would be an enormous number of frequent closed sequences, which would take too much time and memory to compute. On the other hand, when the minimum support threshold is too high, the average length of the frequent closed sequences quickly gets shorter, which in turn negatively affects the selectivity estimation accuracy. In this experiment, our goal is to test the effect of minimum support threshold on the total number of produced patterns and accuracy, and investigate how high the minimum support could be without sacrificing estimation accuracy considerably. Figure 3 shows the number of the frequent closed sequences for different minimum support values.
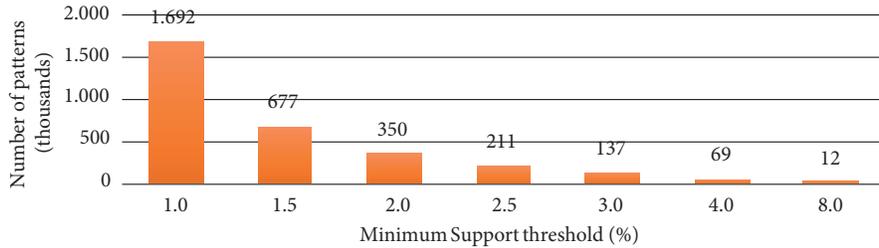
Figure 3. The number of the frequent closed sequences for different min-sup values.
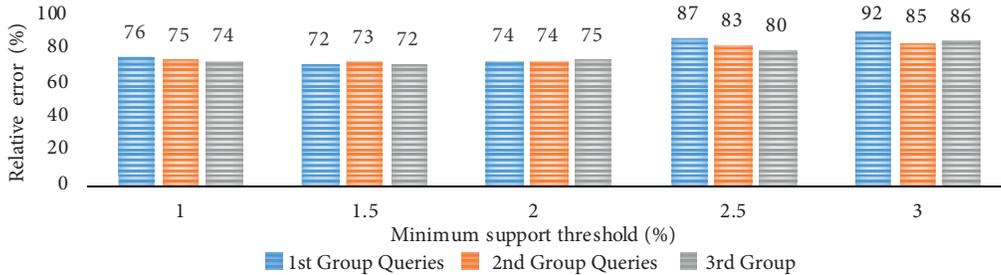
Figure 4. Relative error for different minimum support thresholds values.

Observation 1. *The number of the frequent closed sequences is decreasing significantly to the extent that even 1% change in the minimum support threshold leads to more than 4 times less frequent closed sequences.*

For each minimum support threshold value, we mined the patterns and constructed a histogram based on these patterns. Then, with the constructed histogram, we estimated the selectivity of test query predicates. Figure 4 shows the average relative error for each minimum support threshold value. For all the histograms constructed in this section, the number of buckets is kept constant as 2048 (the effect of different bucket number choices is evaluated in a separate section).

Observation 2. *Having a higher minimum support threshold leads to less frequent closed sequences but does not have a significant effect on the relative error for the interval up until (and including) 2% . Besides, the 2% minimum support threshold case achieves a similar accuracy with about 5 times fewer patterns than the 1% minimum support threshold case. However, when the minimum support threshold becomes greater than 2%, the relative error increases significantly. Hence, for the best accuracy/overhead ratio, 2% may be used as the suggested threshold. If accuracy is the primary focus, then the min-sup value may be slightly decreased to 1.5% while still saving on pattern mining performance, as the number of patterns decreases 2.5 times.*

In this work, we focus on accuracy. Thus, we use min-sup = 1.5% for the rest of the experiments.

5.3.2. The effect of the number of buckets

The number of buckets allowed in a histogram is an important setting for large production databases, as it directly affects the utilization of database kernel cache and the system performance. Ideally, a lower number of buckets is preferred for minimum space and time overhead. However, the accuracy may be hurt with histograms that have a small number of buckets. In this section, we investigate the relationship between the number of buckets allowed in a histogram and the accuracy of selectivity estimation. Figure 5 shows the relative error for different numbers of buckets.
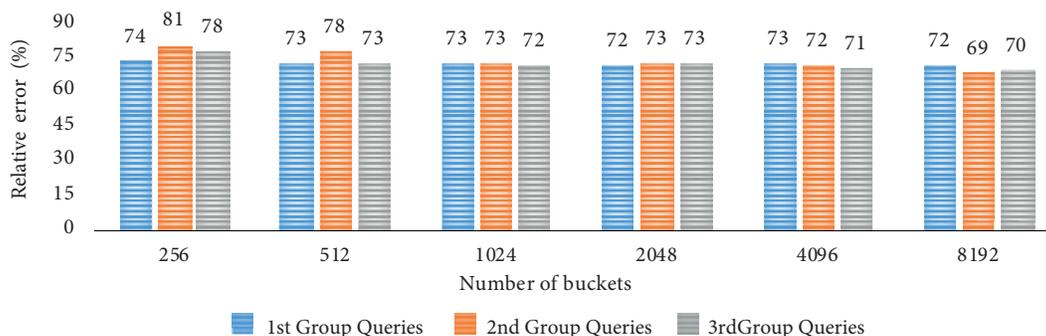
Figure 5. Relative error for different numbers of buckets.

Observation 3. *The increase in the number of buckets does not have a significant effect on the relative error. However, if the number of buckets is smaller than 1024, the accuracy is affected to some extent. This is consistent with the observed decrease in the number of exactly matched and encapsulated matched LIKE query predicates when the bucket number is less than 1024. Hence, 1024 buckets seems to be a reasonable choice to achieve good performance with minimal overhead on the database cache.*

5.3.3. Running time and space overhead

In this section, we evaluate the running time and space overhead of the proposed approach. To this end, we conduct two experiments. In the first experiment, we evaluate the time and space overhead for the offline building phase (i.e. time to mine frequent closed sequences + build a histogram). Figures 6 and 13 show the total build phase time and memory requirements, respectively.
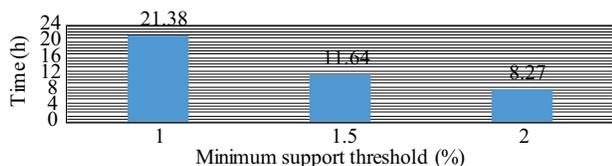

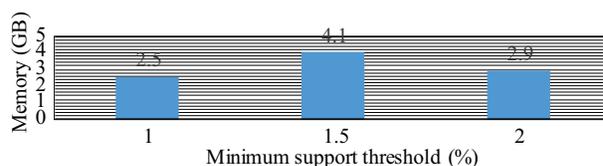**Figure 6**. Build phase time at different minimum support thresholds with 2048 buckets.

Figure 7. Build phase space overhead for both algorithms.

Observation 4. *Build phase imposes significant time and memory overhead, as shown in the above figures. However, since build phase happens offline (before query time) and probably on a different server then the production database, one may tolerate this overhead. Besides, as shown in the next section, the proposed approach is still comparable to the state of the art in terms of build phase requirements.*

In the second experiment, we investigate the time and memory requirements for the online phase (i.e. during query optimization time). Figure 8 shows the average query time for different numbers of buckets at a minimum support threshold of 1.5%. Figure 9 shows the memory requirements for the same experimental setting.
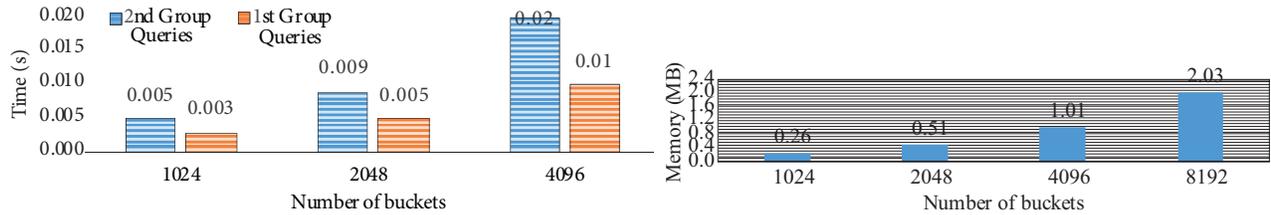

Figure 8. The average query time for both query types with varying numbers of buckets.

Figure 9. Space overhead for different numbers of buckets.

Observation 5. *During query optimization, running time and memory (or catalog) space requirements of SPH are quite low. Hence, it does not add much overhead to the current query optimizer processing.*

5.4. Comparison to the state of art

The state-of-the-art technique in this field is [2]. In this section, we compare our technique (i.e. SPH) to LBS in terms of selectivity estimation accuracy, running time, and memory overhead. Since the source code and test queries are not provided to us by the authors of LBS, we had to implement LBS and the associated [2] and MO [4] algorithms, as well as the algorithm to build the suffix tree [1] ourselves based on the provided details in the corresponding papers. As a sanity check, we generate a test query set using the same approach as described in the LBS paper, and our own implementation of LBS produced very similar accuracy results to those reported in the original LBS paper. For SPH, we set the minimum support threshold as 1.5% and the allowed bucket number as 2048.

5.4.1. Accuracy

In this section, we compare the accuracy of selectivity estimation for LBS and SPH. For LBS, we use the best accuracy-providing settings as reported in the original paper. LBS sets the maximum length of the n-grams to 6 and uses first minima to get the best accuracy for LIKE queries. We run both algorithms on all three groups of test queries. Figure 10 shows the relative error for both algorithms.

Observation 6. *The LBS algorithm, to some extent, is superior to SPH for the 1st group queries, which have the form %w% or %w1%w2%, where wi represents a word with length of at least 5. The difference between the relative errors of LBS and SPH is not more than 23% for this group of queries. On the other hand, for the more generic 2nd group of queries, which have the form $\%s_1\%s_2\%.....\%s_i\%$, where s_i represents one or more characters, SPH is over an order of magnitude more accurate than LBS. The difference between the relative errors of LBS and SPH is more than 1900%. For the query set (3rd group of queries) that is a balanced mix both categories of queries, SPH is greatly superior to LBS (by a margin of 1200%).*

The reason why LBS fails greatly for more generic LIKE queries (i.e. 2nd and 3rd groups of queries) is that LBS inherently assumes a relatively small number of row matches for base substrings. However, for group 2 and 3 queries, base substrings are shorter and may match the majority of rows in the database. In such cases,

LBS's signature computation scheme to estimate the size of the intersection of row ids for base substrings fails with a large error rate.

We next investigate the accuracy of LBS and SPH for negative queries. Figure 11 compares the absolute errors of both approaches for the negative query set.
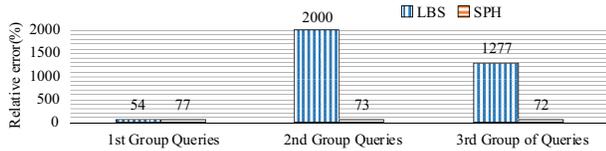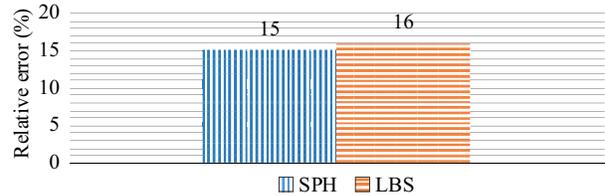

Figure 10. Accuracy comparison of LBS and SPH.

Figure 11. Absolute error for the negative query set for SPH and LBS.

Observation 7. *LBS and SPH provide comparable accuracy for the negative query set.*

5.4.2. Space and time overhead

In this section, we compare the time and space overhead of both algorithms, LBS and SPH. We divide time and space overhead of both algorithms into two separate parts. In the first, we compare time and space overhead of the building phase for both algorithms (i.e. histogram building for SPH and N-gram table building for LBS). Figures 12 and 13 compare time and space overhead of both algorithms, respectively, for the building phase. In the second part, we measure time and space overhead during query optimization time for both algorithms. Figures 14 and 15 compare time and space overhead of both algorithms, respectively, during query optimization time.

Observation 8. *For the offline part (i.e. the building phase), SPH is superior to LBS by a great margin. LBS takes almost three times more time than SPH for the build phase.*

Observation 9. *SPH spends 4 times more memory during the building phase. Since this phase is offline and takes place on a separate server than the production database, this memory overhead may still be tolerable. During the build phase, SPH generates almost 700K closed frequent patterns, while LBS only generates about 94K n-grams when $n \leq 6$. Also, as more efficient sequence pattern mining algorithms are developed, this overhead may significantly decrease in the near future.*
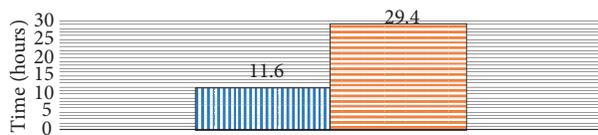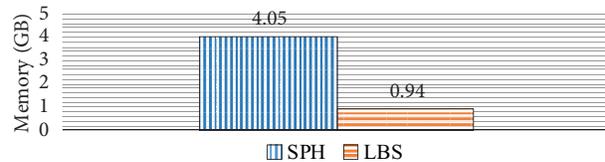

Figure 12. Build phase time overhead for SPH and LBS.

Figure 13. Build phase space overhead for SPH and LBS.

Observation 10. *During query optimization, SPH is more than 7 times faster than LBS.*

Observation 11. *During query optimization time, SPH requires two orders of magnitude less space than LBS. SPH stores the endpoint number, endpoint value, and endpoint frequency values for each bucket in a histogram, while LBS stores pruned suffix tree and all n-grams. Therefore, the histogram structure requires less space than the n-gram table and the pruned suffix tree.*
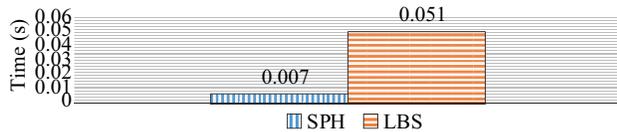
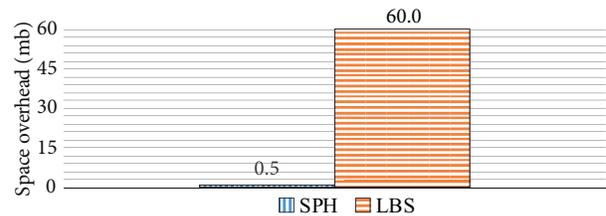**Figure 14**. Average query optimization phase time for SPH and LBS.

Figure 15. Average query optimization phase memory for SPH and LBS.

In this approach, we propose a new technique, SPH, to estimate the selectivity of LIKE query predicates based on a novel summary structure called pattern-based histograms. We build a special histogram structure on top of the sequence patterns extracted from a string database. Then we estimate the selectivity of a LIKE query predicate by employing the histogram structure. We compare SPH to the state of the art and show that it outperforms the state of the art in the accuracy of selectivity estimates for generic LIKE query predicates. Furthermore, it also has significantly lower time and space overhead during query optimization time.

6. Conclusion and future work

In this paper, we propose a new technique, SPH, to estimate the selectivity of LIKE query predicates based on a novel summary structure called pattern-based histograms. We build a special histogram structure on top of the sequence patterns extracted from a string database. Then we estimate the selectivity of a LIKE query predicate by employing the histogram structure. We compare SPH to the state of the art and show that it outperforms the state of the art in the accuracy of selectivity estimates for generic LIKE query predicates. Furthermore, it also has significantly lower time and space overhead during query optimization time.

The sequence patterns discovered by the state-of-the-art sequence mining algorithms do not distinguish consecutive items that always appear together one after another from those items between which there may be others in the database. For instance, let ABB be a frequent pattern discovered by a sequence mining algorithm. It is not possible to figure out if A and B occur together with no other item in between in all the occurrences of ABB in the database. As part of our future work, we plan to mine frequent closed sequences with their relative positions. This may help us to find the selectivity estimation for more specialized LIKE query predicates such as AB, where no characters are allowed between A and B.

Acknowledgment

References

[1] Krishnan P, Vitter JS, Iyer B. Estimating alphanumeric selectivity in the presence of wildcards. In: ACM 1996 Conference on Management of Data; 4–6 June 1996; Montreal, Canada. New York, NY, USA: ACM. pp. 282-293.

[2] Lee H, Ng RT, Shim K. Approximate substring selectivity estimation. In: ACM 2009 International Conference on Extending Database Technology; 24-26 March 2009; Saint Petersburg, Russia. New York, NY, USA: ACM. pp. 827-838.

[3] Jagadish HV, Kapitskaia O, Ng RT, Srivastava D. One-dimensional and multi-dimensional substring selectivity estimation. VLDB J 2000; 9: 214-230.

[4] Jagadish HV, Ng RT, Srivastava D. Substring selectivity estimation. In: ACM 1999 Symposium on Principles of Database Systems; 31 May–2 June 1999; Philadelphia, Pennsylvania, USA. New York, NY, USA: ACM. pp. 249-260.

[5] Chaudhuri S, Ganti V, Gravano L. Selectivity estimation for string predicates: overcoming the underestimation problem. In: IEEE 2004 Conference on Data Engineering; 30 March–2 April 2004; Boston, MA, USA. New York, NY, USA: IEEE. pp. 227-238.

[6] Ferragina P, Grossi R. The string B-tree: a new data structure for string search in external memory and its applications. J ACM 1999; 46: 236-280.

[7] To H, Chiang K, Shahabi C. Entropy-based histograms for selectivity estimation. In: ACM 2013 Conference on Information and Knowledge Management; 27 October–1 November 2013; San Francisco, CA, USA. New York, NY, USA: ACM. pp. 1939-1948

[8] Poosala V, Haas PJ, Ioannidis YE, Shekita EJ. Improved histograms for selectivity estimation of range predicates. In: ACM 1996 Conference on Management of Data; 4–6 June 1996; Montreal, Canada. New York, NY, USA: ACM. pp. 294-305.

[9] Jagadish HV, Koudas N, Muthukrishnan S, Poosala V, Sevcik KC, Suel T. Optimal histograms with quality guarantees. In: VLDB 1998 Conference on Very Large Databases; 24–27 August 1998; New York City, NY, USA. pp. 24-27.

[10] Matias Y, Vitter JS, Wang M. Wavelet-based histograms for selectivity estimation. In: ACM 1998 Conference on Management of Data; 2–4 June 1998; Seattle, WA, USA. New York, NY, USA: ACM. pp. 448-459.

[11] Muralikrishna M, DeWitt DJ. Equi-depth multidimensional histograms. In: ACM 1988 Conference on Management of Data; 1–3 June 1988; Chicago, IL, USA. New York, NY, USA: ACM. pp. 28-36.

[12] Jin L, Li C. Selectivity estimation for fuzzy string predicates in large data sets. In: VLDB 2005 Conference on Very Large Databases; 30 August–2 September 2005; Trondheim, Norway. pp. 397-408

[13] Arasu A, Ganti V, Kaushik R. Efficient exact set-similarity joins. In: VLDB 2006 Conference on Very Large Databases; 12–15 September 2006; Seoul, Korea. pp. 918-929.

[14] Chaudhuri S, Ganjam K, Ganti V, Motwani R. Robust and efficient fuzzy match for online data cleaning. In: ACM 2003 Conference on Management of Data; 9–12 June 2003; San Diego, CA, USA. New York, NY, USA: ACM. pp. 313-324.

[15] Chaudhuri S, Ganti V, Kaushik R. A primitive operator for similarity joins in data cleaning. In: IEEE 2006 Conference on Data Engineering; 3–7 April 2006; Atlanta, GA, USA. New York, NY, USA: IEEE. pp. 5-5.

[16] Xiao C, Wang W, Lin X. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. In: VLDB 2008 Conference on Very Large Databases; 23–28 August 2008; Auckland, New Zealand. pp. 933-944.

[17] Lee H, Ng RT, Shim K. Extending q-grams to estimate selectivity of string matching with low edit distance. In: VLDB 2007 Conference on Very Large Databases; 23–27 September 2007; Vienna, Austria. pp. 195-206.

[18] Kim Y, Park H, Shim K, Woo KG. Efficient processing of substring match queries with inverted variable-length gram indexes. Inf Sci 2013; 244: 119-141.

[19] Yao B, Li F, Hadjieleftheriou M, Hou K. Approximate string search in spatial databases. In: IEEE 2006 Conference on Data Engineering; 1–6 March 2010; Long Beach, CA, USA. New York, NY, USA: IEEE. pp. 545-556.

[20] Agrawal R, Srikant R. Mining sequential patterns. In: IEEE 1995 Conference on Data Engineering; 6–10 March 1995; Taipei, Taiwan. New York, NY, USA: IEEE. pp. 3-14.

[21] Yan X, Han J, Afshar R. CloSpan: Mining: Closed sequential patterns in large datasets. In: SIAM 2003 Conference on Data Mining; 1–3 May 2003; San Francisco, CA, USA. pp. 166-177.

[22] Wang J, Han J, Pei J. Closet+: Searching for the best strategies for mining frequent closed itemsets. In: ACM 2003 Conference on Knowledge Discovery and Data Mining; 24–27 August 2003; Washington, DC, USA. New York, NY, USA: ACM. pp. 236-245.

[23] Zaki MJ, Hsiao CJ. CHARM: An efficient algorithm for closed itemset mining. In: SIAM 2002 Conference on Data Mining; 11–13 April 2002; Arlington, VA, USA. pp. 457-473.

[24] Pasquier N, Bastide Y, Taouil R, Lakhal L. Discovering frequent closed itemsets for association rules. In: ICDT 1999 Conference on Database Theory; 10–12 January 1999; Jerusalem, Israel. Berlin, Germany: Springer. pp. 398-416.

[25] Wang J, Han J. BIDE: Efficient mining of frequent closed sequences. In: IEEE 2004 Conference on Data Engineering; 30 March–2 April 2004; Boston, MA, USA. New York, NY, USA: IEEE. pp. 79-90.