Received July 19, 2020, accepted August 12, 2020, date of publication September 7, 2020, date of current version September 24, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3021945

Supporting Uncertain Predicates in DBMS Using Approximate String Matching and Probabilistic Databases

AMOL S. JUMDE AND RAVINDRA B. KESKAR
Department of Computer Science and Engineering, Visvesvaraya National Institute of Technology, Nagpur 440010, India
Corresponding author: Amol S. Jumde (amol.jumde@students.vnit.ac.in)

ABSTRACT Current relational database systems are deterministic in nature and lack the support for approximate matching. The result of approximate matching would be the tuples annotated with the percentage of similarity but the existing relational database system can not process these similarity scores further. In this paper, we propose a system to support approximate matching in the DBMS field. We introduce a '$\approx$' (uncertain predicate operator) for approximate matching and devise a novel formula to calculate the similarity scores. Instead of returning an empty answer set in case of no match, our system gives ranked results thereby providing a glance at existing tuples closely matching with the queried literals. Two variants of the '$\approx$' operator are also introduced for numeric data: '$\approx+$' for *higher-the-better* and '$\approx-$' for *lower-the-better* cases. Efficient approximate string matching methods are proposed for matching string-type data whereas numeric closeness is used for other types of data (date, time, and number). We also provide results of our system taken over several sample queries that illustrate the significance of our system. All experiments are performed using the MySQL database, whereas the IMDb movie database and European Football database are used as sample datasets.

INDEX TERMS Approximate string matching, probabilistic databases, uncertain predicate.

I. INTRODUCTION

In traditional databases, *select*, *from*, and *where* are the fundamental clauses of any SQL query. General SQL query takes relations specified in the *from* clause as an input, removes tuples which do not satisfy the predicates in the *where* clause, and selects the attributes specified in the *select* clause. In the end, the query returns a filtered relation as an output.

A. PATTERN MATCHING IN DETERMINISTIC DATABASE

Pattern matching in a deterministic database is performed using the *like* operator. Patterns are described using two special characters '%' and '_'. The '%' matches any string *i.e.*, any number of characters and '_' matches a single character. SQL uses the *like* operator to express the pattern. Consider the following query in Fig. 1. This query returns the title of books containing 'Computer' as a substring.

```
select book_title
from books
where book_title like '%Computer%';
```

FIGURE 1. ***like* operator in traditional database.**

Conditions in the *where* clause are used to compare the expressions. The expression can be a column of the relation, or a constant such as string literal (also called as text literal or text constant), a number, date, or an alphanumeric string. We can use comparison operators $<$, $\leq$, $>$, $\geq$, $=$, and $<>$ as well as *like* operator in the *where* clause. Clearly, the traditional database does not support the approximate matching of the values. It can only match the pattern specified in the *like* operator, which is equivalent to the substring matching. The user needs to have comprehensive knowledge about the data in the database, otherwise, the query may return an empty result-set. The same can also happen due to a single misspelled word, or a typing error. Moreover, problems may arise due to the homophonic nature of the words. For example, Cristopher or Kristopher *i.e.* words pronounced in the same way but differing in spellings. If strings are different even in a single character, exact string matching fails. Thus, one of the problems in the existing deterministic database is the lack

The associate editor coordinating the review of this manuscript and approving it for publication was Huanqing Wang.

of support for approximate matching. In this paper, we try to address this issue.

B. NEED OF UNCERTAIN MATCHES

Instead of returning an empty result-set due to above-mentioned reasons like lack of knowledge about the existing data, misspellings, typing mistakes, etc, it will be better to return the result-set ranked according to the degree of similarity. This would help the casual user who has very little knowledge about the stored data. From the approximate results that the user would get, she would come to know about the actual values in the database and she may re-query with the exact keywords that she intended.

Let's assume there exists an *uncertain predicate operator*, denoted as '$\approx$', which is used for approximate matching just like '$=$' operator for exact matching. If somehow approximate matching is performed, resulting rows would have a percentage of matching, where a higher percentage will indicate higher similarity. In such a case, the result of the given query could be a relation annotated with the percentage of matching. These matching scores can be treated as the probability scores or the confidence scores for the respective tuples. Probability scores of filtered tuples need to be stored along with the tuples. The deterministic database has no support for managing probabilities of tuples during query processing. In that direction, the probabilistic database has been proposed by Dalvi and Suciu [1] for managing the probability of a tuple during query processing. Moreover, the queries with *uncertain predicate* using a probabilistic database also have been proposed. For a given SQL query, each tuple in an input database is assigned a probability based on its match with constants in uncertain predicates. Thus the probability for each intermediate tuple is derived. Finally, the result-set is ranked according to its derived probability, indicating the most credible (*i.e.* the highest probable) answer first. For string matching purposes, *Approximate String Matching* (ASM) methods can be used which are discussed in the next section.

C. EXAMPLE OF UNCERTAIN PREDICATE

Our general aim is that using Approximate String Matching(ASM) techniques and probabilistic databases, we should be able to execute the kind of query as shown in Fig. 2.

select book_title, book_author, price
from books
where book_title *like* '%Computer%'
and book_author $\approx$ 'Christopher'
and price $\approx$ 400;

FIGURE 2. Uncertain predicate for approximate matching.

As mentioned, the '$\approx$' operator is the proposed uncertain predicate operator. The expected result is depicted in Table 1 which shows the result-set arranged in the decreasing order of the tuples' probability, topmost tuple being the most similar.

TABLE 1. Expected result of the query shown in Fig. 2.

book_title	book_author	price	prob
Computer Graphics	Christopher	410	0.92
Foundation of Computer Science	Cristopher	360	0.62
Advanced Computer Architecture	Kristopher	455	0.46
Computer Vision	Christopher	315	0.32
⋮	⋮	⋮	⋮

D. OUTLINE AND CONTRIBUTIONS

As per our knowledge, database systems still do not have support for uncertain predicates and it is incompetent to perform approximate string matching. In this work, we extended the idea of queries with uncertain predicates introduced in [1]. In this paper, we try to provide the design of a complete system supporting the uncertain predicates in the database, which, we believe, will make database fields more informative. This will also enable the capabilities of information retrieval to be applied to the database fields. With a probabilistic database in hand, support for uncertain predicates is attainable.

The rest of the paper is organized as follows. We discuss the related work in Section II. In Section III, we take a brief overview of the concepts of the probabilistic database and its use for modeling the uncertainty. We also review various existing ASM methods and propose the best method of interest. Later, we cover the main contributions of the paper:

- We propose the design and implementation of a system to support uncertain predicates along with the regular "certain" predicates.
- We identified the deficiency of basic q-gram distance in our scenario and proposed the global q-gram distance. We provide the complexity and overhead analysis of the proposed formula (Section IV).
- We present the queries to calculate the above-mentioned distance by utilizing the in-built features provided by the DBMS (Section IV).
- We also introduce two variants ($\approx +$ and $\approx -$) of uncertain predicate operator ($\approx$) for the numeric type of data (Section IV).
- We formulate the procedures to implement our proposed method. We also describe details of the sample databases that we have used for the testing (Section V).
- We demonstrate the results of several queries on sample databases (Section VI).

Finally, we conclude in Section VII with possible future directions.

II. RELATED WORK

SQL query with uncertain predicate is first proposed by Dalvi and Suciu [1]. But, the main focus of their work was on query evaluation in probabilistic databases. They have given intensional and extensional query evaluation to derive the probability of the filtered tuples. But, to the best of our knowledge, a full-fledged system supporting uncertain predicate is still missing. Probabilistic databases have been widely explored for storing uncertain data, instead of just replacing uncertain fields with *nulls* [2]. Hassanzadeh and Miller [3] have used probabilistic databases in order to remove duplicate records from the base dirty relation. They find out the similarity between the records and use clustering algorithms to identify the set of records that are probable duplicates. Kanagal *et al.* [4] proposed the sensitivity analysis to find the most influential input tuples for the query which predominantly affects the probability of the answer. Qin and Yu [5] proposed the dynamic programming algorithm to perform sensitive analysis for Inequality Query (IQ) efficiently. Sensitivity analysis can be utilized for ranking the query results [6].

Hakak *et al.* [7] presented a comprehensive survey on exact string matching algorithms that are commonly used for pattern searching. Unlike exact string matching, Approximate String Matching(ASM) methods allow errors in matching, which makes them useful for applications like spell checker, autocompletion, etc [8], [9]. ASM methods are also used to extract features for machine learning and to calculate intra-comment similarity to tackle the issue of cybercrimes like cyberbullying, online aggression, etc, over the social media [10]–[13]. Loo [14] has included *stringdist* package in R for approximate string matching. Methods in *stringdist* package have an optional parameter to specify a string matching method to be used among several well known implemented methods. Parallel versions of ASM methods using Graphics Processing Units (GPUs) are discussed in [15].

Another area of DBMS that finds the application of ASM is the approximate string join [16], [17]. Effective pruning techniques are inevitable in this area. Various filters based on the count filtering, position filtering, length filtering [18], subtrie pruning [19], minimal-edit-distance pruning, duplication pruning [20] have been proposed to improve the performance.

The probabilistic database system has promising applications in various fields that have to deal with uncertain, imprecise, or inaccurate information [21], [22]. But, a concrete probabilistic database management system is still lacking. Only a few university projects like MayBMS [23]–[25], Trio [26] have been proposed.

With the proposal of probabilistic database, additional queries like top-k [27]–[29], skyline queries [30], [31] over probabilistic databases, are proposed in the literature. Both of these queries highlight the dominating tuples from the database. Top-k and Skyline queries are very useful in the fields of decision making and multi-criteria analysis. In probabilistic databases, tuples or attributes are assigned with some probability. Therefore, the probabilistic database exists in multiple instances and each instance is known as '*possible world*'. If possible world semantics is followed for aggregate queries, a query may return an exponentially sized result-set, *i.e.*, one aggregate result for each possible world. In order to overcome this problem, three variants of aggregate queries have been proposed by Murthy *et al.* [32]. These are termed as low bound, high bound, and the expected value of the aggregate values.

III. PRELIMINARIES

In this section, we briefly take an overview of the probabilistic database and ASM methods which are the building blocks of our system.

A. UNCERTAINTY AND PROBABILISTIC DATABASES

To deal with uncertain data and probabilistic query processing, *probabilistic databases* are proposed [33]. Nowadays, many applications have to deal with uncertain data, such as object recognition, sensor networks, moving object monitoring, data extraction, and integration [21]. Uncertainty may arise during the data collection, data transmission, or data processing stages.

With traditional deterministic databases, uncertainty cannot be modeled. The use of *nulls* is proposed to handle the uncertainty by Codd [34]. But, there can be an information loss in using the *nulls* in place of doubtful values. For example, consider a classifier that classifies the objects. It could identify that object X belongs to class A with 0.6 probability and class B with 0.4 probability. Due to this uncertainty, if the class of X is recorded as *null*, the information that we had earlier is lost. Such type of uncertainty can be best modeled and processed using probabilistic databases [35]–[39]. It stores uncertainty in terms of a probability for that uncertain tuple or attribute. The probability of a tuple or an attribute is a measure of confidence that the system has on that reading.

Formally, probabilistic databases can be defined as follows:

Definition 3.1 (Probabilistic Database and Possible World [23]): Consider a schema with relations $R_1, \ldots, R_k$. A *probabilistic database* is a finite set of structures $W = \{W^1, W^2, \ldots, W^n\}$, where each $W^i = \langle R_1^i, \ldots, R_k^i, p^{[i]} \rangle$, is called a *possible world* and its probability of existence is $p^{[i]}$.

Further, $\sum_{1 \leq i \leq n} p^{[i]} = 1$. Every relation R_j^i in world W^i contains one of the possible subsets of tuples from relation R_j. If relation R_j contains only the *certain tuples i.e.*, tuples with probability 1, it is called as *certain relation*, and $R_j^1 = R_j^2 = \cdots = R_j^n = R_j$.

Let us consider the following example of probabilistic database.

1) EXAMPLE OF PROBABILISTIC DATABASE

Consider, an institute has installed an automated system for maintaining students' attendance. The system recognizes a student using facial recognition with some error. The percentage of the match can be treated as a probability of that tuple. One particular instance of such a table is given in Table 2. The database with such uncertain relations is called a *probabilistic database*.

2) POSSIBLE WORLD SEMANTICS

As seen in the previous example, every tuple is assigned with some probability of existence. For instance, tuple t_1 is true with 0.9 probability and at the same time, it is false with

TABLE 2. Example of probabilistic database.

	Record_No	Name	Time	Location	conf
t_1	2018-11-17-001	Rakesh Kumar	15:09:02	CSE-101	0.9
t_2	2018-11-17-002	Ankur Bhagat	15:11:34	CSE-101	0.8
t_3	2018-11-17-003	Roshni Thakur	15:12:21	CSE-102	0.85
t_4	2018-11-17-004	Neetu Singh	15:15:51	CSE-101	1.0
	⋮	⋮	⋮	⋮	⋮

0.1 probability. Thus, in one instance, some of the tuples will remain true and the rest will be false. This forms one possible instance of the above database. In this way, the probabilistic database can exist in multiple possible instances. In one of the possible instances, all tuples will remain true. Similarly, in one particular instance, a table may not have any tuple (this can happen only if there is no certain tuple in a relation). Such multiple instances are called *possible worlds* of a probabilistic database.

A traditional relational database has only one state, containing all the present tuples. In contrast, a probabilistic database can be in one of several states. It is a set of multiple possible certain databases each of which has some chance to appear in the real world. The probability of a possible world is determined by the probabilities of its tuples. For example, probability of possible world pw containing three tuples (say t_1, t_4, and t_5) out of five tuples is, $P_{pw}\{t_1, t_4, t_5\} = p_1 * (1 - p_2) * (1 - p_3) * p_4 * p_5$, where p_1, p_4, p_5 are the probabilities of presence of tuples t_1, t_4, and t_5, respectively, and $(1 - p_2)$ and $(1 - p_3)$ represent the probabilities of the absence of tuples t_2 and t_3.

B. APPROXIMATE STRING MATCHING

Approximate String Matching (ASM) is a string matching problem where two strings are matched approximately rather than exactly. ASM aims to find the closest match for a query string. The uncommon part between the two strings is considered as the error between them. As ASM allows errors, it is useful in cases of undesired corruption or inadvertent typing errors. Most often, the approximate string match is measured in terms of distance. Exactly matched strings will have a distance of 0. ASM distance metrics are classified into three major categories, namely *edit based*, *q-grams based* (sometimes called as n-grams), and *heuristic based* [9], [14].

The *edit based* metrics are measured in terms of the number of edit operations required to make two strings equal. These edit operations include addition, deletion, and substitution of a character, or transposition of characters. Based on the allowed operations, edit based distances are categorized as Hamming distance [40], Longest Common Subsequence (LCS) distance [41], Levenshtein (LV) distance [42], Optimal String Alignment (OSA) distance, and Damerau-Levenshtein (DL) distance [43].

The *q-gram based* distances aim to find common q-grams between two matched strings. The q-grams of string s is a set of all possible q length substrings of s. The maximum number of distinct q-grams for string s is $|s| - q + 1$. For example, if $q = 2$ and $s =$ 'VNIT' then the q-grams are {'VN', 'NI', 'IT'}. Jaccard distance, q-gram distance [44], and Cosine distance are some examples of q-gram based distances.

Definition 3.2 (q-Gram Distance): The *q-gram distance* is the number of q-grams that are not common between the two strings and can be calculated as:

$$d_{qgram}(s, t; q) = \|\upsilon(s; q) - \upsilon(t; q)\|_1 = \sum_{i=1}^{|\Omega|^q} |\upsilon_i(s; q) - \upsilon_i(t; q)|$$

where, Ω represents the finite set of alphabets.[1] The Kleene closure set (Ω^*) is an infinite set of all possible strings over Ω. The set Ω^q represents all the strings of length q from Ω^*. The vector $\upsilon(s; q)$ is a q-gram profile of string s of dimension $|\Omega|^q$ whose coefficients represent the number of occurrences of every possible q-grams of length q. Let, a q-gram $x \in \Omega^q$, has an index i in the vector $\upsilon(s; q)$ then the $\upsilon_i(s; q)$ represents the total occurrences of x in s. Thus, the q-gram distance is equal to the L_1-norm of vectors of two strings [44]. It varies between 0 to $|s| + |t| - 2q + 2$. For example, the q-gram distance between 'Tuesday' and 'Thursday' is 7 for $q = 2$ as 'Tu', 'ue', 'es', 'Th', 'hu', 'ur', and 'rs' are the q-grams that are not common between them.

Lastly, *heuristic based* distances have a particular application in mind, but do not have a solid mathematical backbone. For example, Jaro distance [45] was specifically designed to pair the records between the census data and the independent Post Enumeration Survey (PES) conducted by the U.S. Census Bureau in 1985. The intuition behind the method is that the types of errors like transpositions of characters or character mismatches between close characters are more likely due to the typing errors like martha-marhta or jonathon-jonathn. These are legitimate variations and should have lesser distances. It is evident from the following examples:

d_{jaro}('martha', 'marhta') $= 0.05555556$
d_{jaro}('martha', 'artham') $= 0.1111111$
d_{jaro}('martha', 'megan') $= 0.4222222$

Though there is a single transposition of characters in the first two examples, the Jaro distance of the first example is less as it may be more likely due to the typing error.

After this brief overview of uncertainty and probabilistic databases, as well as Approximate String Matching (ASM), our proposed methodology based on these two concepts is described in the next section.

IV. METHODOLOGY

In order to find a similarity between the values in terms of distance, we introduce '$\approx$' operator for the uncertain predicate. To implement such an operator, for string type data, we can use the distance between the strings which is also a notion of dissimilarity between them. For numerals, the distance could be the indicator of numeric closeness. Fig. 3 shows the basic modules of the proposed system which are *preprocessing*,

[1] In the literature, the Σ symbol is generally used to represent the finite set of alphabets. But here we used the Ω symbol to avoid the confusion between the set of alphabets and the summation operation.

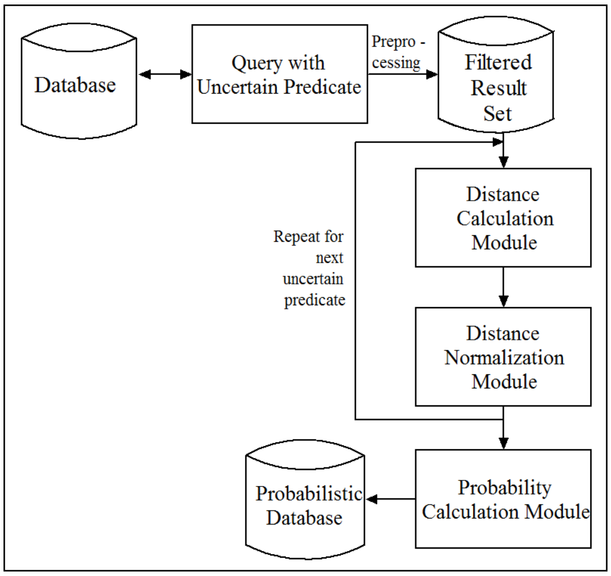

Database
Query with Uncertain Predicate
Prepro - cessing
Filtered Result Set
Distance Calculation Module
Repeat for next uncertain predicate
Distance Normalization Module
Probabilistic Database
Probability Calculation Module

FIGURE 3. Block diagram of the proposed system.

distance calculation, *distance normalization*, and *probability calculation*. When the parser encounters '≈' symbol in the *where* clause of the query, it performs approximate matching on the columns involved in uncertain predicates. A query may have other predicates along with the uncertain predicate, as present in our previous *books* example in Section I (again depicted in Fig. 4(a)). An instance of the *books* relation is shown in Fig. 4(b). All predicates, except uncertain predicates, are applied first and then uncertain predicates are applied on those filtered result-set. These steps are the preprocessing steps of our system. Fig. 4(c) illustrates the filtered result-set. *Distance Calculation Module* calculates the distance between the queried literal and each of the field values in the corresponding column to get the distance array (Fig. 4(d)). Distance array is normalized in the range [0, 1] by the *Distance Normalization Module* (Fig. 4(e)). The above steps are repeated for all the uncertain predicates. *Probability Calculation Module* combines the probabilities obtained from the uncertain predicates to calculate the final probability of filtered tuples (Fig. 4(f)) and finally, we get a probabilistic database as an output (Fig. 4(g)).

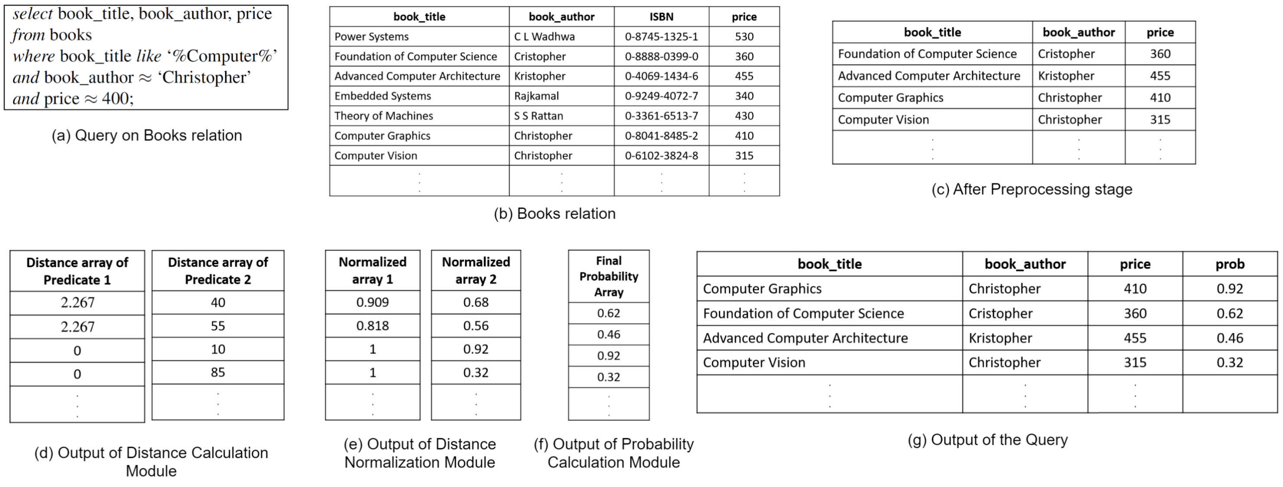
Procedures to find distances differ based on the data type of a column. Most of the standard databases support various built-in data types, such as *number* for integer type and real type data, *char* for a character, *varchar* for a string. Along with these basic types, databases also support types for a date (*date*) and time (*time*).

Firstly, we focus on string-type data. Out of all the ASM techniques discussed in the previous section, we believe q-gram distance (with required *modifications*) is best suited for our purpose. Edit based distances do not fare well in case of swapped words cases. For example, 'John Watson' may get recorded as 'Watson, John'. The edit based distance between these two strings is 10 (for OSA) out of a possible range of [0, 12], which is significant. In the case of q-gram distance, it would be just 5 (for q = 2) out of a possible range of [0, 21]. We have neglected heuristic-based methods as they are very much application-specific.

In Section IV-A we describe a naive solution to find the distance between two strings. In Section IV-B we propose a more accurate distance formula that overcomes the drawbacks of the naive solution.

A. NAIVE SOLUTION TO CALCULATE THE DISTANCE BETWEEN TWO STRINGS

As noted earlier, q-gram distance needs $O(|\Omega|^q)$ storage. To avoid such high space complexity, we use the following technique to calculate the q-gram distance between the two strings (s and t) in RDBMS. Here one can try to utilize in-built features already available with the RDBMS like efficient joins methods, clustered index, etc. Auxiliary tables, $A_1(qgram)$ and $A_2(qgram)$, with $qgram$ as the only attribute,



```
select book_title, book_author, price
from books
where book_title like '%Computer%'
and book_author ≈ 'Christopher'
and price ≈ 400;
```

(a) Query on Books relation

book_title	book_author	ISBN	price
Power Systems	C L Wadhwa	0-8745-1325-1	530
Foundation of Computer Science	Cristopher	0-8888-0399-0	360
Advanced Computer Architecture	Kristopher	0-4069-1434-6	455
Embedded Systems	Rajkamal	0-9249-4072-7	340
Theory of Machines	S S Rattan	0-3361-6513-7	430
Computer Graphics	Christopher	0-8041-8485-2	410
Computer Vision	Christopher	0-6102-3824-8	315
⋮	⋮	⋮	⋮

(b) Books relation

book_title	book_author	price
Foundation of Computer Science	Cristopher	360
Advanced Computer Architecture	Kristopher	455
Computer Graphics	Christopher	410
Computer Vision	Christopher	315
⋮	⋮	⋮

(c) After Preprocessing stage

Distance array of Predicate 1	Distance array of Predicate 2
2.267	40
2.267	55
0	10
0	85
⋮	⋮

(d) Output of Distance Calculation Module

Normalized array 1	Normalized array 2
0.909	0.68
0.818	0.56
1	0.92
1	0.32
⋮	⋮

(e) Output of Distance Normalization Module

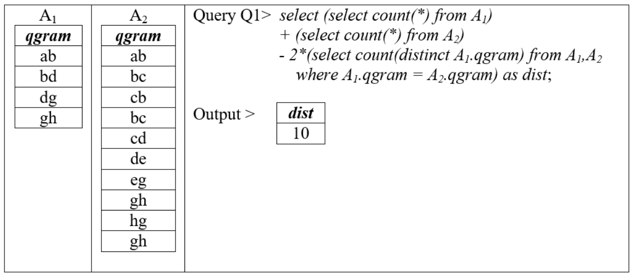
Final Probability Array
0.62
0.46
0.92
0.32
⋮

(f) Output of Probability Calculation Module

book_title	book_author	price	prob
Computer Graphics	Christopher	410	0.92
Foundation of Computer Science	Cristopher	360	0.62
Advanced Computer Architecture	Kristopher	455	0.46
Computer Vision	Christopher	315	0.32
⋮	⋮	⋮	

(g) Output of the Query

FIGURE 4. Working of the proposed system on the *books* relation example.



A_1

qgram
ab
bd
dg
gh

A_2

qgram
ab
bc
cb
bc
cd
de
eg
gh
hg
gh

```
Query Q1> select (select count(*) from A1)
          + (select count(*) from A2)
          - 2*(select count(distinct A1.qgram) from A1,A2
            where A1.qgram = A2.qgram) as dist;

Output >
```

dist
10

FIGURE 5. Naive solution using auxiliary tables.

are created on-the-fly respectively for the two strings and deleted upon use. The example given in Fig. 5 depicts auxiliary tables A_1 and A_2 tables storing q-grams of s = 'abdgh' and t = 'abcbcdeghgh' strings. Query Q1 in Fig. 5 is the SQL query to find the q-gram distance between the strings *s* and *t*.

B. PROPOSED SOLUTION TO CALCULATE THE DISTANCE BETWEEN TWO STRINGS

The problem in using q-gram distance in its purest form is that the user may query a part of the original string present in the table. For example, she may query 'Harry Potter' instead of 'Harry Potter and the Order of the Phoenix' while searching for the movie titles. Such extra uncommon parts of one of the strings increase the distance. To overcome this problem, we introduce the concepts of *Local q-gram distance* and *Global q-gram distance*. We define them as follows:

Definition 4.1 (Local q-Gram Distance): The *local q-gram distance* is the q-gram distance between the smaller length string and one of the substrings of the same length of the larger string.

The larger string (say y), among the two strings to be matched, is divided into the set of substrings (y_i*'s*) where length of each substring is equal to the length of smaller-length string (say x). Let, l_x, l_y be the length of strings x and y, respectively. Let, $n = l_y - l_x + 1$, which will be the number of possible substrings (y_i*'s*) of length l_x of a string y. The local q-gram distances (d_i) between x and each of y_i's are calculated using query Q1. The proposed net distance between the strings x and y is derived from their local q-gram distances to be termed as 'Global q-gram distance'.

Definition 4.2 (Global q-Gram Distance): The *global q-gram distance* between strings x and y is proposed to be calculated as:

$$d(x, y) = min(d_1, d_2, \ldots, d_n) + \frac{l_y - l_x}{l_x} + \frac{\sum_{i=1}^{n} d_i}{maxdist(x, y)}$$

Here $d_1, d_2, \ldots, d_n$ are the local q-gram distances between two strings calculated as mentioned above. The next two terms are the fine-tuning terms that help to calculate a more accurate distance. A *maxdist(x,y)* is the sum of maximum distance possible between the x and each of y_i's and therefore $maxdist(x, y) = 2n(l_x - q + 1)$.

Now we explain the significance of each term in the global q-gram distance formula proposed above. The overall aim is to reduce the increased distance between two strings due to the missing part of the query string, the lacuna in the naive solution mentioned earlier. We first note that the first string may be contained anywhere in the second string. Hence, to minimize the distance between two strings, minimum of local q-gram distances is considered ($min(d_1, d_2, \ldots, d_n)$) as the first term in the formula to calculate the global q-gram distance. But, it can be quickly seen that this is not enough. Even if *d('aaa', 'aaa')* is 0 and *d('aaa', 'aaaaa')* is also 0 (for q = 2), still, the first string should get precedence over the second in terms of similarity. Hence, there is a need for a penalty for the difference in the relative lengths of the strings, which is computed by $(l_y - l_x)/l_x$ as the second term in the formula. Hence, higher the length differences (amongst the matched strings), higher will be the penalty. Now a penalty with a fractional value (0.667) gets added in the calculation of *d('aaa', 'aaaaa')* as against *d('aaa', 'aaa')* which remains at 0. Even then, *d('aaa', 'aaaaa')* is 0.667 and the *d('aaa', 'baaad')* is also 0.667. Here we believe that '*aaa*' is more closer to '*aaaaa*' than '*baaad*' as '*aaa*' gets matched multiple times in '*aaaaa*' than in '*baaad*'. To factor in this observation, the third term ($\sum_{i=1}^{n} d_i/maxdist(x, y)$) in the formula is added in the calculation of global q-gram distance. It denotes the ratio of summation of actual distances of the substrings ($\sum_{i=1}^{n} d_i$) to the maximum possible distance between the strings x and y ($maxdist(x, y)$). This ratio measures the amount of cumulative dissimilarity between the two strings measured relatively against the maximum distance. Distance is maximum when there is no common character or common q-gram between the strings x and y. In the example discussed above, the sum of distances with respect to '*baaad*' ($0 + 0.667 + 0.333 = 1$) is more as against '*aaaaa*' ($0 + 0.667 + 0 = 0.667$) due to this third term, and hence '*aaaaa*' will be considered as a better match of '*aaa*' compared to '*baaad*', as required.

Table 3 depicts some sample strings and their distances when the formula is applied in a part-wise manner. The distances calculated using the global distance formula in the table justifies the significance of each term of the formula.

TABLE 3. Result of proposed formula on some sample strings.

String 1	String 2	Considering only the 1st term of the formula	Considering term 1 & 2 of the formula	Considering term 1, 2 & 3 of the formula
aaaa	aaaa	0	0	0
aaaa	aaaaaa	0	0.5	0.5
aaaa	baaaac	0	0.5	0.7222
aaaa	aaaabc	0	0.5	0.8333
aaaa	aaabc	2	2.25	2.75
aaaa	aaa aaa	2	2.75	3.25
aaaa	abcde	6	6.25	7.25

Let AUX be an auxiliary table used to store local q-gram distances. To calculate the global q-gram distance, we execute query Q2 in Fig. 6 on the AUX table. Before executing Q2,

AUX
dist
d1
d2
...
dn

```
>SET @q = 2
>SET @lx = LENGTH(s1);
> SET @ly = LENGTH(s2);
> SET @maxdist = 2*(@ly-@lx+1)*(@lx-@q +1)

Query Q2> select (select min(dist) from AUX)
        + (select (@ly-@lx)/@lx)
        + ((select sum(dist) from AUX) / @maxdist);
```

FIGURE 6. Auxiliary table storing local q-gram distances and SQL query for calculating the global q-gram distance.

we need to set constants like q, l_x, l_y and *maxdist* using MySQL commands as shown in Fig. 6.

For other data types like numeric or date, a simple absolute difference of the values would be the distance between them. For example, a unit of distance between two date values is the number of days separating them, like d('1973-12-30', '1973-10-15') = 76 days. In this way, approximate matching is performed for all the columns involved in uncertain predicates and corresponding distance arrays are obtained.

C. COMPLEXITY ANALYSIS

In case of a naive solution, the q-gram distance between the strings x and y can be computed in $O(|x| . |y|)$ time and the memory requirement is $O(|Q(y; q)|)$ storage, where $Q(y; q)$ denotes a set of q-grams existing in string y, if $l_y > l_x$.

Overhead due the creation of substrings in global q-gram distance will increase the time complexity to $O(|x|^2 . |y|)$ but, memory requirement will reduce to $O(|Q(x; q)|)$. This overhead comes with improved accuracy of the system.

D. VARIANTS OF '≈' OPERATOR FOR NUMERIC DATA

Along with '≈' operator, we introduce '≈+' and '≈−' operators. The operator '≈+' indicates the *higher-the-better* case. Sometimes the higher distance between the values is beneficial. For example, if we are searching for books with discount nearly equal to 20%. It is better to show books with higher discount at the top. Similarly, '≈−' indicates the *lower-the-better* case. In our book example, we were searching for books with a price nearly equal to 400. Here, it is better to show books with lower prices at the top. For '≈' operator we take the absolute difference between the values. But in the case of '≈+', we need to take the specific difference (constant - column_value) and in the case of '≈−', the (column_value - constant).

E. NORMALIZATION OF DISTANCE ARRAY

All distance arrays may differ in the range of values. For the uniformity, distance arrays are normalized in the range [0, 1] using the following formula to obtain the probability array:

$$p[\,] = 1 - \frac{dist[\,] - min}{max - min}$$

The probability of a match is equal to one minus the normalized score, because $d = 0$ indicates the exact match, whereas, in probability notion, $p = 1$ indicates the exact match. When we get the distance array for data type other than string, we assume the highest element in an array to be the maximum calculated distance (*max*). In this way, the probability array for each of the columns involved in uncertain predicates is calculated.

F. FINAL PROBABILITY CALCULATION

A query may have multiple uncertain predicates combined with *and* and *or* operators. Moreover, each uncertain predicate is an independent probabilistic event, which means the occurrence of one predicate will not affect the probability of other predicates. Therefore, the conditional probability of a predicate A, in the presence of predicate B, is equal to the probability of the predicate A *i.e.*, $p(A|B) = p(A)$. Also, the precedence of *and* operator is higher than the *or* operator. Therefore, the probability of 'n' predicates combined with *and* operator is first calculated using the following formula:

$$p = (p_1 \cap p_2 \cap \cdots \cap p_n) = p_1 * p_2 * \cdots * p_n$$

The probability for 'm' predicates combined by *or* operator is calculated by using the principle of Inclusion-Exclusion as given below:

$$\begin{aligned} p = (p_1 \cup p_2 \cup \cdots \cup p_m) &= \sum_{1 \leqslant i \leqslant m} p_i \\ &- \sum_{1 \leqslant i_1 < i_2 \leqslant m} (p_{i_1} \cap p_{i_2}) + \sum_{1 \leqslant i_1 < i_2 < i_3 \leqslant m} (p_{i_1} \cap p_{i_2} \cap p_{i_3}) \\ &- \cdots + (-1)^{m-1} (p_1 \cap \cdots \cap p_m) \end{aligned}$$

In case of a complex expression containing multiple operators like *and* and *or*, operations are performed based on the operator precedence. Parenthesis can be used to override the actual precedence of operators. For nested parenthesis, innermost parenthesis is evaluated first.

For example, consider a query with four uncertain predicates combined with one *and* and two *or* operators applied sequentially without explicitly resolving the precedence. After performing approximate matching, let 0.7, 0.6, 0.4, and 0.5 be the probabilities of the corresponding four columns for the resultant tuple t_1. Then, the final probability of tuple t_1 is $p(t_1) = 0.7 \cap 0.6 \cup 0.4 \cup 0.5 = (0.7 \cap 0.6) \cup 0.4 \cup 0.5 = 0.42 \cup 0.4 \cup 0.5 = 0.826$.

In the next section, we summarize the entire methodology in the form of set of procedures. We discuss the implementation details and the datasets used for the experimentation, as well.

V. IMPLEMENTATION, DATASETS AND EXPERIMENTATION

To support uncertain predicate operator '≈', the query parser should recognize newly introduced '≈' operator. For experimentation, we parsed the input query in the C code itself to check whether the given query has '≈' operator. If the input query has a '≈' operator, we first filter out tuples by executing a query with certain predicates. Afterward, we apply uncertain predicates on those filtered result-set by invoking procedure *UncertainPredicate* given in Procedure 1.

Procedure 1 UNCERTAINPREDICATE(*uncertain-query*)

```
1:  Let, $m$ be a number of filtered tuples after applying
    predicates except uncertain predicates
2:  Retrieve columns over which uncertain predicate is specified and let $n$ be the number of those columns
3:  Let, $dist[1 \ldots m, 1 \ldots n]$ be a matrix to store distances of
    'm' tuples for 'n' attributes
4:  for $i$ =1 to $n$ do
5:      Let, $Constant_i$ be a constant involved in uncertain
        predicate of $Column_i$
6:      if typeOf($Column_i$) = 'varchar' then
7:          $(dist[:, i])$ =
            CALCULATEDISTANCEFORSTRING($Column_i$, $Constant_i$)
8:      else
9:          $(dist[:, i])$ =
            CALCULATEDISTANCEFOROTHER($Column_i$, $Constant_i$)
10:     $(dist[:, i])$ = NORMALIZEDISTANCE($dist[:, i]$)
11: Let, $prob[1 \ldots m]$ be the array to store probabilities of $m$
    tuples
12: $prob[1 \ldots m]$ = DERIVEPROBABILITY($dist$)
13: return $prob$
```

Procedure 2 CALCULATEDISTANCEFORSTRING(*Column*, *Constant*)

```
1: Let, $d[1 \ldots Column.length]$ be the array for storing calculated distances
2: $q$ = getValueOfQ()
3: for $j$ =1 to $Column.length$ do
4:     $d[j]$ = calGlobalQgramDist($Column[j]$, $Constant$, $q$)
5: return $d$
```

Procedure 3 CALCULATEDISTANCEFOROTHER(*Column*, *Constant*)

```
1:  Let, $d[1 \ldots Column.length]$ be the array for storing calculated distances
2:  if typeOf($Column$) = 'date' then
3:      for $j$ =1 to $Column.length$ do
4:          $d[j]$ = calDaysDifference($Column[j]$, $Constant$)
5:  else if typeOf($Column$) = 'time' then
6:      for $j$ =1 to $Column.length$ do
7:          $d[j]$ = calMinsDifference($Column[j]$, $Constant$)
8:  else if typeOf($Column$) = 'number' then
9:      for $j$ =1 to $Column.length$ do
10:         $d[j]$ = calDifference($Column[j]$, $Constant$)
11: return $d$
```

Distance between the constant mentioned in the query and the corresponding column field is calculated based on the data type of the column. Type of a column is determined using *typeOf()* function. Procedure *CalculateDistanceForString()* is used for the column of string type. For other data types, *CalculateDistanceForOther()* procedure is called.

Matrix $dist[m][n]$ stores the calculated distance for m tuples and n columns. *NormalizeDistance()* function normalizes the distance array of every column in [0-1] range. *DeriveProbability()* derives the probability of tuples using probability inference rules discussed in the previous section.

In Procedure 2, the value of 'q' is obtained using *getValueOfQ()* function. Using *calGlobalQgramDist()* function, we calculate the global q-gram distance. It issues query Q2 to the database which itself makes use of query Q1.

In Procedure 3, *calDaysDifference()* function is used to calculate the distance in days, for a column of type date. Distance for the time data type is calculated using *calMinsDifference()* in the units of minutes. For the numeric data type, distance is the difference of values and it is calculated using *calDifference()* function.

For experimentation, we have used the MySQL database. MySQL C API library is used to communicate with MySQL server. The *libmysqlclient* is a client version of the library that C applications use to communicate. This C API offers several data structures for database connection, for storing the result of a query, like, MYSQL, MYSQL_RES, MYSQL_ROW. It also provides various C API functions like mysql_init() for initializing MYSQL structure, mysql_real_connect() for connecting to the MySQL server, mysql_query() for executing the query and many more [46].

A. DATABASES USED FOR TESTING

IMDb movie [47] and European football [48] databases were used for testing. IMDb movie database has information about 5000+ movies. IMDb database has only one relation named 'Movies'. Original relation had too many attributes related to the movies. In our version of 'Movies' relation, we inserted only 11 important attributes amongst them. These are Movie_Title, Director_Name, Duration, Actor_1_Name, Actor_2_Name, Actor_3_Name, Gross, Genres, Budget, Title_Year, and IMDb_Score.

Football database contains information of 11000+ players and their attributes, 25000+ matches, and 11 leagues. 'Player' relation stores personal information about the player and his attributes values are stored in 'Player_Attributes' relation. Attributes in 'Player' relation are id, player_api_id, player_name, player_fifa_api_id, birthday, height, and weight. 'Player_Attributes' relation has 42 columns that cover all attributes of a football player. player_api_id and player_fifa_api_id attributes are the foreign keys referring to the identical attributes in the 'Player' relations. Some fields in these databases were missing or incorrect. Our modified versions of these databases are available at [49].

VI. RESULTS AND DISCUSSION

In this section, we present the results of different experiments. As part of the experimentation, we compared the performances of the naive approach with the proposed solution.

We also compared the results of the proposed global q-gram distance with the classic edit based distances. We analyzed the effect of the value of q on accuracy. We have also presented the results of additionally proposed operators ($\approx+$ and $\approx-$).

select Movie_Title
from Movies
where Movie_Title ≈ 'X-Men Origins';

FIGURE 7. Sample query on the IMDb database.

A. NAIVE SOLUTION VS. PROPOSED SOLUTION

Here we compare the performance of the naive solution as mentioned in Section IV-A with the proposed solution in Section IV-B. One sample query on the IMDb movie database is shown in Fig. 7. Table 4 shows their results on some random strings. Queried strings and the expected strings are shown in the first two columns of the table. The third and fourth columns of the table represent the result of a naive approach and the proposed technique, respectively. A naive method fails when there is a significant mismatch between the lengths of two strings.

TABLE 4. Results of naive method and the proposed method (q = 2).

	Queried Strings	Expected Strings	Result of naive method	Result of proposed method
ex1	X-Men Origins	X-men Origins:Wolverine	I Origins	X-men Origins:Wolverine
ex2	Life of Pets	The Secret Life of Pets	Life of Pi	The Secret Life of Pets
ex3	Most Wanted	A Most Wanted Man	Wanted	A Most Wanted Man
ex4	Twilight Saga Moon	The Twilight Saga: New Moon	Moonlight Mile	Twilight
ex5	Karate Kid	The Karate Kid	The Karate Kid	The Karate Kid

In the proposed method, there is an overhead of matching individual substrings which resulted in the change of the complexity from $O(|x| \,.\, |y|)$ to $O(|x|^2 \,.\, |y|)$. The actual increase in execution times is depicted in Fig. 8. The Movies relation contains 5043 tuples. Each of the execution times shown in Fig. 8 is the average of 10 experiments. An average of execution times of all the experiments (5*10 = 50 experiments) shows a 46.3% increase in the execution time. But, this trade-off comes with improved accuracy as shown in Table 4 and Fig. 8.

B. EDIT BASED DISTANCE VS. GLOBAL Q-GRAM DISTANCE

As mentioned earlier, we noted that the edit-based distance would fail to identify similarities between the swapped-words strings (*i.e.* 'John Watson' vs 'Watson, John'). Here, we illustrate it further using a few examples. In query 1 (see Fig. 9), we intend to get a film from the Harry Potter film series, which has its IMDb score is near to 7.0 but let's say we (wrongly) suppose, is directed by 'David Bates'. In reality, some parts of this film series are directed by David Yates (and not Bates). Tables 5-7 show the correct part of the movies at the top of the tables. We tried OSA and LCS methods from the edit-based category. We obtained the intended results for all the methods, as shown in the tables mentioned above.

In query 2 (see Fig. 10), we have given the case of swapped words. Here we queried for titles which are similar

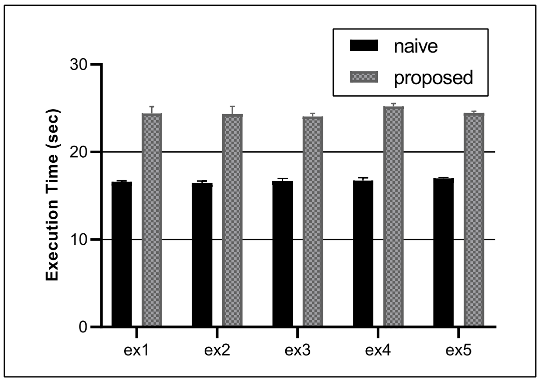


FIGURE 8. Execution times of the two techniques on the sample strings.

select Movie_Title, Director_Name, IMDb_Score
from Movies
where Movie_Title ≈ 'Harry Potter and the '
and Director_Name ≈ 'David Bates'
and IMDb_Score ≈ 7.0;

FIGURE 9. Query 1 on the IMDb database.

TABLE 5. Result of Query 1 using the global q-gram method (q = 2).

Movie_Title	Director_Name	IMDb_Score	prob
Harry Potter and the Half-Blood Prince	David Yates	7.5	0.720278827
Harry Potter and the Order of the Phoenix	David Yates	7.5	0.716742963
Fury	David Ayer	7.6	0.432688066
Home Run	David Boyd	6	0.379173609
Dune	David Lynch	6.6	0.366126543
⋮	⋮	⋮	⋮

TABLE 6. Result of Query 1 using the OSA method.

Movie_Title	Director_Name	IMDb_Score	prob
Harry Potter and the Half-Blood Prince	David Yates	7.5	0.772979095
Harry Potter and the Order of the Phoenix	David Yates	7.5	0.768858737
Redbelt	David Mamet	6.8	0.419750558
Astro Boy	David Bowers	6.3	0.394052863
Harsh Times	David Ayer	7	0.377957241
⋮	⋮	⋮	⋮

TABLE 7. Result of Query 1 using the LCS method.

Movie_Title	Director_Name	IMDb_Score	prob
Harry Potter and the Half-Blood Prince	David Yates	7.5	0.790239782
Harry Potter and the Order of the Phoenix	David Yates	7.5	0.78708994
Hard Candy	David Slade	7.1	0.461035402
Fury	David Ayer	7.6	0.458849522
Redbelt	David Mamet	6.8	0.4306562
⋮	⋮	⋮	⋮

```
select Movie_Title, Actor_1_Name, Actor_2_Name,
    Actor_3_Name
from Movies
where Movie_Title ≈ 'Hunters of Ghost'
and ( Actor_1_Name ≈ 'Steve'
or Actor_2_Name ≈ 'Steve'
or Actor_3_Name ≈ 'Steve' );
```

FIGURE 10. Query 2 on the IMDb database.

to 'Hunters of Ghost' in which an actor named 'Steve' has acted. But the real name of the movie is 'Ghost Hunters' and the complete name of the actor is 'Steve Gonsalves'. IMDb database has three columns for storing the names of three actors and we are not sure which column of actor names contains 'Steve'. Notice that we have used parenthesis to override the default precedence of the operators and to associate these three conditions. In the q-gram distance, a set of all possible q-grams is formed. So, inherently, the sequence of the substrings does not affect the distance. Thus, it is expected that the q-gram method would give the intended result at the top. On the other hand, OSA and LCS perform sequential matching. Hence, even if two strings have exact same substrings and if one has them swapped, distance is not zero. For example, d_{qgram}('Ghost Hunters', 'Hunters of Ghost', q = 2) = 7 out of a possible range of [0-27], d_{osa}('Ghost Hunters', 'Hunters of Ghost') = 14 out of a possible range of [0-16] [Operations: SSSDMIIIMSSSSSSSMI], and d_{lcs}('Ghost Hunters', 'Hunters of Ghost') = 15 [Operations: DDDDDDMMMMMMMIIIIIIII], where 'S' stands for substitution, 'D' for deletion, 'I' for Insertion, and 'M' for match of the character. Results of Query 2 on the IMDb database are as shown in Tables 8-10.

TABLE 8. Result of Query 2 using the global q-gram method (q = 2).

Movie_Title	Actor_1_Name	Actor_2_Name	Actor_3_Name	prob
Ghost Hunters	Amy Bruni	Steve Gonsalves	Jason Hawes	0.602986823
Khumba	Liam Neeson	Steve Buscemi	Loretta Devine	0.564523379
Minions	Steve Carell	Jon Hamm	Steve Coogan	0.556277755
Ghost	Demi Moore	Tony Goldwyn	Phil Leeds	0.533126168
Ghost World	Scarlett Johansson	Steve Buscemi	T.J. Thyne	0.532182172
⋮	⋮	⋮	⋮	⋮

C. ADDITIONAL OPERATORS

To illustrate the utility of the proposed uncertain operator '≈', two additional operators ('≈+' and '≈−') are also proposed as an extension. We conducted several experiments to test the practicability of the proposed operators. We executed the same query first with '≈' and then using '≈+/≈−' operators and examined the results. In query 3 (see Fig. 11), we tried to utilize our system as a predictor. We want to find the best striker from the Football database. There are several attributes that represent the player's skills and his/her performance. Key attributes of any striker are finishing, dribbling, acceleration, composure, and pace. To get the name of the players, we joined 'Player' and 'Player_Attributes' relations on the 'player_api_id' attribute. Table 11 shows the result of Query 3 which has today's leading strikers at the top.

TABLE 9. Result of Query 2 using the OSA method.

Movie_Title	Actor_1_Name	Actor_2_Name	Actor_3_Name	prob
Diner	Steve Guttenberg	Daniel Stern	Ellen Barkin	0.583522671
Yentl	Miriam Margolyes	Amy Irving	Steven Hill	0.579984826
Ghost	Demi Moore	Tony Goldwyn	Phil Leeds	0.57555179
Good	Viggo Mortensen	Jodie Whittaker	Steven Mackintosh	0.570624901
Minions	Steve Carell	Jon Hamm	Steve Coogan	0.559427524
⋮	⋮	⋮	⋮	⋮

TABLE 10. Result of Query 2 using the LCS method.

Movie_Title	Actor_1_Name	Actor_2_Name	Actor_3_Name	prob
Diner	Steve Guttenberg	Daniel Stern	Ellen Barkin	0.715666305
Good	Viggo Mortensen	Jodie Whittaker	Steven Mackintosh	0.685238459
Yentl	Miriam Margolyes	Amy Irving	Steven Hill	0.642897913
Wings	Steven Weber	Tim Daly	Amy Yasbeck	0.637444788
Ghost	Demi Moore	Tony Goldwyn	Phil Leeds	0.602204576
⋮	⋮	⋮	⋮	⋮

```
select player_api_id, player_name, finishing,
    acceleration, dribbling
from Player a, Player_Attributes b
where a.player_api_id = b.player_api_id
and finishing ≈ 92
and acceleration ≈ 92
and dribbling ≈ 92;
```

FIGURE 11. Query 3 on the Football database.

TABLE 11. Result of Query 3.

player_api_id	player_name	finishing	acceleration	dribbling	prob
37412	Sergio Aguero	90	92	89	0.954149049
30893	Cristiano Ronaldo	95	91	93	0.953682394
19533	Neymar	88	91	94	0.931503733
30981	Lionel Messi	93	95	96	0.917279953
325916	Paulo Dybala	87	90	89	0.897924639
30834	Arjen Robben	85	90	93	0.897667059
40636	Luis Suarez	90	88	88	0.894923997
⋮	⋮	⋮	⋮	⋮	⋮

In query 4 (see Fig. 12), we demonstrated the use of the '≈+' operator. Players with better finishing, acceleration and dribbling scores should be ranked first. Table 12 shows the result of query 4, where we can observe that Messi now has jumped to the top of the table.

In query 5 (see Fig. 13), we tried to find out the highest-grossing movies. Therefore, the gross amount should be higher than the constant in the condition as much as possible, and the budget should be as low as possible. Values of gross and budget specified in the conditions are the average values of the gross and budgets of all the movies. Table 13 shows the results of Query 5 and indicates Avatar as the highest-grossing movie of that period.

```
select player_api_id, player_name, finishing,
    acceleration, dribbling
from Player a, Player_Attributes b
where a.player_api_id = b.player_api_id
and finishing ≈+ 92
and acceleration ≈+ 92
and dribbling ≈+ 92;
```

FIGURE 12. Query 4 on the Football database.

TABLE 12. Result of Query 4.

player_ api_id	player_name	finishing	acceleration	dribbling	prob
30981	Lionel Messi	93	95	96	0.954968944
30893	Cristiano Ronaldo	95	91	93	0.897619048
19533	Neymar	88	91	94	0.838854382
37412	Sergio Aguero	90	92	89	0.820190649
30834	Arjen Robben	85	90	93	0.789794686
107417	Eden Hazard	81	93	94	0.789510007
325916	Paulo Dybala	87	90	89	0.771859903
⋮	⋮	⋮	⋮	⋮	⋮

```
select Movie_Title, Gross, Budget
from Movies
where Gross ≈+ 51054995.24
and Budget ≈- 39816476.04;
```

FIGURE 13. Query 5 on the IMDb database.

TABLE 13. Result of Query 5.

Movie_Title	Gross	Budget	prob
Avatar	760505847	237000000	0.888602778
Jurassic World	652177271	150000000	0.797095509
Titanic	658672302	200000000	0.784679151
The Avengers	623279547	220000000	0.734811236
The Dark Knight	533316061	185000000	0.640285926
Star Wars: Episode IV - A New Hope	460935665	11000000	0.602957172
⋮	⋮	⋮	⋮

D. EFFECT OF THE VALUE OF q

Our experiments show that a small value of q is more likely to give better results. According to [9], the optimal value of $q = log_{|\Omega|} |s|$, where Ω is the finite set of alphabets. Therefore, the value of $q = 2$ or 3 is more favorable. The results are not reliable for $q = 1$. It is obvious because no q-gram overlapping happens and no ordering of characters is checked when q-gram size is 1.

Unlike traditional database field, *Information Retrieval (IR)* techniques retrieve results based on the matching of a set of keywords from the query string. Returned web pages may not match with all the keywords, but they do match with only the part of it. Moreover, IR techniques always return a result-set sorted according to their percentage of matching. Thus, IR techniques differ from database queries with respect to these two features, first, the approximate matching and second, a ranking of the result-set showing most promising result at the top. By providing support for uncertain predicates in this work, we are, in a way, trying to achieve the same features in traditional databases as well.

VII. CONCLUSION AND FUTURE WORK

In this paper, a database management system to support the uncertain predicates is proposed. The similarity between the values (of strings) is measured in terms of a modified q-gram distances, *i.e.* local q-gram distance and global q-gram distance. A novel formula to calculate these distances between the strings is proposed and explained in detail. A set of sample queries indicate how this could be beneficial. Probabilities of filtered tuples are then meaningfully derived from the normalized distance arrays, which are treated as a confidence score for that tuple. Finally, the result-set is ranked according to the decreasing values of probabilities, indicating the most likely matches at the top. As an extension, we have also introduced two additional variants of the uncertain predicate which add to the practicability of the system. We demonstrated the significance of our proposed system through the various sample queries. We believe that our system will surely be helpful for the users having a minimal or superficial knowledge of the actual existing data in the database. The proposed global q-gram distance may find its application in other fields like record linkage, data deduplication, DNA sequence alignment.

The output of our proposed system is a probabilistic relation. The derived column named '*prob*' is augmented to the result-set as an additional column. A concrete system that could be useful on a wider scale can be implemented by incorporating all proposed procedures in an open-source database system. For further query processing, query evaluation techniques of probabilistic databases can as well be utilized for processing uncertain relations.

REFERENCES

[1] N. Dalvi and D. Suciu, "Efficient query evaluation on probabilistic databases," *VLDB J.*, vol. 16, no. 4, pp. 523–544, Aug. 2007.
[2] D. Suciu, D. Olteanu, C. Ré, and C. Koch, "Probabilistic databases," *Synth. Lectures Data Manage.*, vol. 3, no. 2, pp. 1–180, 2011.
[3] O. Hassanzadeh and R. J. Miller, "Creating probabilistic databases from duplicated data," *VLDB J.*, vol. 18, no. 5, pp. 1141–1166, Oct. 2009.
[4] B. Kanagal, J. Li, and A. Deshpande, "Sensitivity analysis and explanations for robust query evaluation in probabilistic databases," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2011, pp. 841–852.
[5] B. Qin and J. X. Yu, "Efficient sensitivity analysis for inequality queries in probabilistic databases," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 1, pp. 86–99, Jan. 2017.
[6] J. Chen, Y. Li, and L. Feng, "Sensitivity analysis of answer ordering from probabilistic databases," in *Proc. Int. Conf. Database Expert Syst. Appl.* Berlin, Germany: Springer, 2013, pp. 243–258.
[7] S. I. Hakak, A. Kamsin, P. Shivakumara, G. A. Gilkar, W. Z. Khan, and M. Imran, "Exact string matching algorithms: Survey, issues, and future research directions," *IEEE Access*, vol. 7, pp. 69614–69637, 2019.
[8] P. Jokinen, J. Tarhio, and E. Ukkonen, "A comparison of approximate string matching algorithms," *Softw., Pract. Exper.*, vol. 26, no. 12, pp. 1439–1458, 1996.
[9] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, Mar. 2001.
[10] Y. Li, Y. Peng, W. Ji, Z. Zhang, and Q. Xu, "User identification based on display names across online social networks," *IEEE Access*, vol. 5, pp. 17342–17353, 2017.

[11] D. Chatzakou, N. Kourtellis, J. Blackburn, E. De Cristofaro, G. Stringhini, and A. Vakali, "Mean birds: Detecting aggression and bullying on Twitter," in *Proc. ACM Web Sci. Conf. (WebSci)*, 2017, pp. 13–22.
[12] D. Chatzakou, N. Kourtellis, J. Blackburn, E. De Cristofaro, G. Stringhini, and A. Vakali, "Measuring #Gamergate: A tale of hate, sexism, and bullying," in *Proc. 26th Int. Conf. World Wide Web Companion*, 2017, pp. 1285–1290.
[13] D. Chatzakou, N. Kourtellis, J. Blackburn, E. De Cristofaro, G. Stringhini, and A. Vakali, "Hate is not binary: Studying abusive behavior of #Gamergate on Twitter," in *Proc. 28th ACM Conf. Hypertext Social Media*, 2017, pp. 65–74.
[14] M. P. Van der Loo, "The stringdist package for approximate string matching," *R J.*, vol. 6, no. 1, pp. 111–122, 2014.
[15] T. Ho, S.-R. Oh, and H. Kim, "A parallel approximate string matching under levenshtein distance on graphics processing units using warp-shuffle operations," *PLoS ONE*, vol. 12, no. 10, Oct. 2017, Art. no. e0186251.
[16] J. Jestes, F. Li, Z. Yan, and K. Yi, "Probabilistic string similarity joins," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2010, pp. 327–338.
[17] J. Lu, C. Lin, W. Wang, C. Li, and H. Wang, "String similarity measures and joins with synonyms," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2013, pp. 373–384.
[18] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava, "Approximate string joins in a database (almost) for free," in *Proc. VLDB*, vol. 1, 2001, pp. 491–500.
[19] J. Wang, J. Feng, and G. Li, "Trie-join: Efficient trie-based string similarity joins with edit-distance constraints," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 1219–1230, Sep. 2010.
[20] J. Wang, G. Li, and J. Fe, "Fast-join: An efficient method for fuzzy token matching based string similarity join," in *Proc. IEEE 27th Int. Conf. Data Eng.*, Apr. 2011, pp. 458–469.
[21] M. Ye, W.-C. Lee, D. L. Lee, and X. Liu, "Distributed processing of probabilistic top-k queries in wireless sensor networks," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 1, pp. 76–91, Jan. 2013.
[22] T. Ge, A. Dekhtyar, and J. Goldsmith, "Uncertain data: Representations, query processing, and applications," in *Advances in Probabilistic Databases for Uncertain Information Management*. Berlin, Germany: Springer, 2013, pp. 67–108.
[23] C. Koch, "MayBMS: A system for managing large uncertain and probabilistic databases," in *Managing and Mining Uncertain Data*, vol. 149. Berlin, Germany: Springer-Verlag, 2009.
[24] L. Antova, C. Koch, and D. Olteanu, "MayBMS: Managing incomplete information with probabilistic world-set decompositions," in *Proc. IEEE 23rd Int. Conf. Data Eng.*, Apr. 2007, pp. 1479–1480.
[25] J. Huang, L. Antova, C. Koch, and D. Olteanu, "MayBMS: A probabilistic database management system," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2009, pp. 1071–1074.
[26] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. Nabar, T. Sugihara, and J. Widom, "Trio: A system for data, uncertainty, and lineage," in *Proc. 32nd Int. Conf. Very Large Data Bases (VLDB)*, 2006, pp. 1151–1154.
[27] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang, "Top-k query processing in uncertain databases," in *Proc. IEEE 23rd Int. Conf. Data Eng.*, Apr. 2007, pp. 896–905.
[28] C. Re, N. Dalvi, and D. Suciu, "Efficient top-k query evaluation on probabilistic data," in *Proc. IEEE 23rd Int. Conf. Data Eng.*, Apr. 2007, pp. 886–895.
[29] X. Lian and L. Chen, "Top-k dominating queries in uncertain databases," in *Proc. 12th Int. Conf. Extending Database Technol. Adv. Database Technol. (EDBT)*, 2009, pp. 660–671.
[30] J. Pei, B. Jiang, X. Lin, and Y. Yuan, "Probabilistic skylines on uncertain data," in *Proc. 33rd Int. Conf. Very Large Data Bases*, 2007, pp. 15–26.
[31] X. Liu, D.-N. Yang, M. Ye, and W.-C. Lee, "U-skyline: A new skyline query for uncertain databases," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 4, pp. 945–960, Apr. 2013.
[32] R. Murthy, R. Ikeda, and J. Widom, "Making aggregation work in uncertain and probabilistic databases," *IEEE Trans. Knowl. Data Eng.*, vol. 23, no. 8, pp. 1261–1273, Aug. 2011.
[33] D. Amagata, Y. Sasaki, T. Hara, and S. Nishio, "Probabilistic nearest neighbor query processing on distributed uncertain data," *Distrib. Parallel Databases*, vol. 34, no. 2, pp. 259–287, Jun. 2016.
[34] E. F. Codd, "Extending the database relational model to capture more meaning," *ACM Trans. Database Syst.*, vol. 4, no. 4, pp. 397–434, Dec. 1979.
[35] T. Imieliński and W. Lipski, "Incomplete information in relational databases," *J. ACM*, vol. 31, no. 4, pp. 761–791, Sep. 1984.
[36] N. Dalvi, C. Ré, and D. Suciu, "Probabilistic databases: Diamonds in the dirt," *Commun. ACM*, vol. 52, no. 7, pp. 86–94, Jul. 2009.
[37] N. Fuhr and T. Rölleke, "A probabilistic relational algebra for the integration of information retrieval and database systems," *ACM Trans. Inf. Syst.*, vol. 15, no. 1, pp. 32–66, Jan. 1997.
[38] T. J. Green and V. Tannen, "Models for incomplete and probabilistic information," in *Proc. Int. Conf. Extending Database Technol.* Berlin, Germany: Springer, 2006, pp. 278–296.
[39] A. S. Jumde and N. S. Chaudhari, "Query processing techniques in probabilistic databases," in *Proc. Int. Conf. Comput., Analytics Secur. Trends (CAST)*, Dec. 2016, pp. 483–488.
[40] R. W. Hamming, "Error detecting and error correcting codes," *Bell Syst. Tech. J.*, vol. 29, no. 2, pp. 147–160, Apr. 1950.
[41] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *J. Mol. Biol.*, vol. 48, no. 3, pp. 443–453, Mar. 1970.
[42] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Sov. Phys.-Dokl.*, vol. 10, no. 8, pp. 707–710, 1966.
[43] R. A. Wagner and R. Lowrance, "An extension of the string-to-string correction problem," *J. ACM*, vol. 22, no. 2, pp. 177–183, Apr. 1975.
[44] E. Ukkonen, "Approximate string-matching with q-grams and maximal matches," *Theor. Comput. Sci.*, vol. 92, no. 1, pp. 191–211, Jan. 1992.
[45] M. A. Jaro, "Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida," *J. Amer. Stat. Assoc.*, vol. 84, no. 406, pp. 414–420, Jun. 1989.
[46] M. Widenius, D. Axmark, and M. AB, "Mysql 5.5 reference manual," Oracle Corp., Redwood City, CA, USA, Tech. Rep. 5.5, 2014.
[47] Yueming. (2017). *IMDB 5000 Movie Dataset*. Accessed: Oct. 12, 2019. [Online]. Available: https://www.kaggle.com/carolzhangdc/imdb-5000-movie-dataset
[48] H. Mathien. (2016). *European Soccer Database*. Accessed: Oct. 12, 2019. [Online]. Available: https://www.kaggle.com/hugomathien/soccer
[49] (2017). *Modified Datasets*. Accessed: Oct. 12, 2019. [Online]. Available: https://sites.google.com/students.vnit.ac.in/uncetain-predicate-datasets/home

AMOL S. JUMDE received the B.Tech. degree in computer science and engineering from the Government College of Engineering, Amravati, India, in 2011, and the M.Tech. degree in computer science and engineering from the Walchand College of Engineering, Sangli, India, in 2015. He is currently pursuing the Ph.D. degree in computer science and engineering with the Visvesvaraya National Institute of Technology (VNIT), Nagpur, India.

From 2014 to 2015, he was an Intern with the Center for Research in Engineering Sciences and Technology (CREST), KPIT Technologies Ltd., Pune, India. His research interests include probabilistic databases, machine learning, and image processing.

RAVINDRA B. KESKAR received the B.Tech. degree in computer science and engineering from the Visvesvaraya National Institute of Technology (VNIT), Nagpur, India, in 1997, the M.Tech. degree in computer science and engineering from the Indian Institute of Technology Bombay, in 1999, and the Ph.D. degree in computer science and engineering from VNIT, in 2015.

From 1999 to 2004, he was a Senior Research Associate with Sasken Communication Technologies. From 2004 to 2009, he was a Senior Technical Lead with Persistent Systems Ltd. Since 2009, he has been an Associate Professor with the Department of Computer Science and Engineering, VNIT. His research interests include real-time and distributed systems, machine learning, cloud computing, and compiler optimization.