Survey on String Similarity Joins

[1]Ritika Jaiswal, [2]Priyanka Nikam

Department of Computer Engineering,
VJTI, Mumbai, India

***Abstract* -In case of the databases containing large number of data values in the form of strings, string similarity join operation plays an important role. Similarity join is used in data integration and data cleansing to find similar string pairs from two sets of strings. If the database contains very large number of rows, then such a comparison becomes extremely expensive. There are various algorithms to perform similarity join on strings, but each has its own pros and cons. It becomes very strenuous for the practitioners to decide which algorithm should be used in which scenario. To help practitioners choose a suitable algorithm, we provide a comparative study on the existing string similarity join algorithms. We arrange the algorithms in different classes, based on their methodology. We also mention the strengths and weaknesses of these algorithms, to help practitioners pick an appropriate algorithm.**

***Index Terms* – Similarity Join, data integration, data cleaning.**

I. INTRODUCTION

String similarity join finds all pairs of strings that have some of their portion in common. It is useful in many real-world applications like data cleansing, data integration, pattern matching, duplicate data detection, etc. The similarity between the two strings is decided by similarity functions. Similarity functions can be of two types:

- Set-based similarity functions
- Character-based similarity functions

The examples of set-based similarity functions are Jaccard, Dice, Cosine and for character-based similarity functions, the example is Edit distance.

Set-based metrics are suitable for long strings, e.g., documents. The procedure is to first transform strings into sets of tokens and then quantify their similarity. There are two ways to transform strings into sets: (1) tokenization and (2) q-grams. The first one uses special characters e.g., white-space characters to tokenize strings. The second one divides a string into substrings of length q and generates the set. The substring of length q is called a q-gram.

The character-based similarity metrics quantify the similarity between two strings based on character transformations. They take care of typographical errors. One example of character-based metric is edit distance. Edit distance is the count of the minimum number of operations required to transform one string into the other. It is a way of quantifying how dissimilar two strings (e.g., words) are to one another by counting the minimum number of operations required to transform one string into the other.

Jaccard index, also known as the Jaccard similarity coefficient[1], is a statistic used for comparing the similarity and diversity of sample sets. The Jaccard coefficient measures similarity between finite sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets. Given two strings a & b, and let A and B denote the q-gram(which are tokens of size q), Jaccard coefficient is defined as -

$\text{Jaccard}(A,B) = (A \cap B)/(A \cup B)$

On a similar line, $\text{cosine}(A,B) = (A \cap B)/\sqrt{(|A|.|B|)}$ and $\text{dice} = 2*(A \cap B)/ (|A|+|B|)$

Two strings are similar if the value of their similarity metric is greater than a threshold T.

Majority of existing similarity join methods use in-memory algorithms. These algorithms have a limit on the size of dataset. Since the advent of big-data, the size of datasets have increased by heaps and bounds and therefore there is need for algorithms that can take care of very large datasets.

One way of dealing with this is the brute-force algorithm, that enumerates every string pair and checks whether the two strings in the pair are similar. This is time consuming and a lot expensive.

Another way is to implement a filter-verification framework, which comprises of two steps:

(1) Filter step: This step will rule out a large numbers of dissimilar pairs and generate a set of candidate pairs.

(2) Verification step: In this step, we verify each candidate pair by computing the real similarity and the output the final results.

Filtering algorithms in the first step is very crucial in the framework. A lot of the existing filtering algorithms use a signature-based technique. They generate signatures for each string, and if two strings are similar their signatures will overlap. In this way, those strings which have no common signature are eliminated. Some of the most commonly used filtering techniques are -

- Count filtering- if two strings are similar, their signatures will share at least T common signatures
- Length filtering - if two strings are similar, their length will be nearly equal
- Position filtering - It uses positions of signatures to prune dissimilar pairs. It mandates that two strings must share atleast T matching positional q-grams, to be similar.

- Prefix filtering - It quickly filters out the candidate pairs that are guaranteed not to meet the T threshold.
- Content filtering - The idea is to select a probing window and look into the contents of both strings within the probing window. The content difference in the probing windows, will give us a lower bound on the edit distance of the pair.

Among these, prefix filtering is the most effective filtering technique. Number of algorithms have been proposed to improve prefix filtering for different similarity metrics. Some of these algorithms are-

- AllPair [2]
- EDJoin [3]
- QChunk [4]
- VChunk [5]

In this paper, , we provide a comparative study on the existing string similarity join algorithms. We arrange the algorithms in different classes, based on their methodology. This will help the practitioners pick an appropriate algorithm.

II. OVERVIEW OF ALGORITHMS

In this paper, we concentrate on two prime techniques of filtering - count and prefix filtering. Two existing algorithms apply Count-filtering as the main technique. These are GramCount[6] and ListMerger[7]. The other algorithms AllPair, EdJoin, QChunk and VChunk extend the prefix-filtering idea to implement the string similarity join. The table shown below present on overview of the algorithms, and depicts the prominent differences in their methodology. The deciding parameter for all these algorithms are different. Some are dependent on the size of gram i.e 'q'. Also, the metric used by the algorithms to judge whether two strings are similar differ.

Table 1: Algorithms and supported metrics

Techniques	Algorithms	Parameters	Metrics
Count	GramCount	q (for ED)	All
	ListMerger	q (for ED)	All
Prefix	AllPair	q (for ED)	All
	EDJoin	NA	ED
	QChunk	q	ED
	VChunk	rules	ED

Where,

All=(ED)U(TOKEN)

TOKEN=(JAC,COS,DICE)

III. COUNT FILTERING

L. Gravano et.al. explained count filtering in GramCount [6]. The intuition behind count filtering is that the two strings must share at least T common signatures if they are similar. This means that if two strings share less than T signatures, that string pair should be discarded. GramCount supports edit distance using this property while another algorithm ListMerger[7] improves GramCount so as to support token-based metrics.

Computing T: GramCount takes q-grams as signature by generating q-grams for each string. Let τ be the edit distance threshold. One edit operation prunes at most q grams. String s has $|s| + 1 - q$ grams and if strings r and s are similar, their signature sets share at least $T=\max(|s|,|r|)+1-q-q\tau$ signatures. ListMerger accepts every token as a signature. Let δ be the overlap threshold. Two strings are similar with respect to the overlap similarity, so they must have in common at least δ signatures, and $T = \delta$.

Pruning with T: ListMerger creates inverted indexes for signatures. For each string s, ListMerger produces its signatures and obtains inverted lists of its signatures. For each string r on these lists, ListMerger calculates its count number, that is the number of lists containing r. Obviously if the count number of string r is small than T, the pair (r, s) can be pruned.

IV. PREFIX FILTERING

Prefix filtering enables similarity joins inside DBMS. AllPair[2] enhances this idea and proposes the idea of prefix filtering based framework. Overlap similarity explains how prefix filtering works as illustrated below:

Overlap: Assume that we have ordered all tokens in the string sets as, e.g., the alphabetical order or inverse document frequency (idf) order. The prefix filtering arranges its tokens according to the given token ordering for each string. For each string s, the prefix filtering selects the initial $|s| - \delta + 1$ tokens as the signatures, denoted by s_p, given that the overlap threshold is δ. It is easy to prove that if two strings r and s are similar, then $s_p \cap r_p \neq \varphi$. For example, assume the first two strings "database concept system" and "database mining techniques". Let the overlap threshold be 2. So the prefix size is 3-2+1=2. Assume that the tokens are arranged by idf in the descending order. Prefix filtering selects "concept system" and "mining techniques" as their prefixes. These prefixes do not have any overlap, therefore this pair can be destroyed. The existing methods use idf in the descending order of the tokens and take tokens with large idf as signatures.

Edit Distance: The edit distance is supported by the gram based method. For each string it generates the q-gram set, sorts the grams according to the global ordering, e.g. alphabetical order or idf order, and selects the initial $q\tau + 1$ grams as its signatures. It is easy to prove that if r and s are identical, then their prefixes shares the common signatures. This occurs because each edit operation can prune at most q q-grams, and if more than $q\tau + 1$ grams are pruned, it requires at least $\tau+1$ edit operations. For example, consider the two strings "sigmpo" and "sigir" with the edit-distance threshold $\tau = 1$. It sorts q-grams by using idf in the descending order of the grams and obtains their prefixes (underlined grams): {gm, mp, po, ig, si} and {gi, ir, ig, si}. As no common signature is shared by this pair, it can be pruned.

4.1 Optimizations on Edit Distance

4.1.1. EDJoin : The EDJoin algorithm[3] proposes following optimization techniques to improve prefix filtering for edit distance:

(1) Position Filtering: EDJoin eliminates the unnecessary signatures from the prefix to further decrease the prefix length. For example, consider a string "sigmpo" with $q = 2$ and $\tau = 1$. Consider that the signatures are "si", "gm" and "po" in the prefix filtering. EDJoin proves that it can eliminate the last signature "po" safely since it requires at least $\tau + 1$ edit operations to prune both "si" and "gm". EDJoin devises effective algorithms to detect and remove the signatures that are not required.

(2) Content Filtering: The position filtering destroys the pairs that are not similar using positions of the grams. It is efficient for mismatch grams distributed in different positions in the string but inefficient for consecutive grams. To mitigate this problem, EDJoin proposes content filtering method to do further pruning. For example, consider the following two strings: sigmpd and sigbdd. Let $\tau = 1$ be the edit-distance threshold. As the two substrings are rather dissimilar, EDJoin can destroy this pair. To achieve this goal, EDJoin takes the L1 distance as a bound of edit distance for pruning, where L1 distance between two strings is the number of different characters in the two strings. EDJoin proves that the edit distance is always greater than or equal to half of the L1 distance. For example, the L1 distance of the two substrings is 6 so the edit distance between the two strings is at least 3. Therefore, it can be pruned.

4.1.2. QChunk: The QChunk algorithm[4] consists of two types of signatures: q-grams and q-chunks. The q-chunks are q-grams with starting positions at $i * q + 1$ for $0 \leq i \leq (l-1)/q$ where l is the string length. To guarantee that the last q-chunk has exactly q characters, it adds several special characters, e.g., #. For example, the q-chunks of "vldaj" are {"vl", "da", "j#"}. Suppose g_q denote the q-gram set and c_q denote the q-chunk set. QChunk constructs two strategies to destroy the pairs that are not similar. The first one is to index the q-grams of string r and use q-chunks of string s to obtain candidates. The second one is to index q-chunks of string r and use q-grams of string s to obtain candidates. Both of the two strategies can be used into prefix filtering framework. Moreover, QChunk describes efficient techniques to reduce the number of signatures to lower bound $\tau + 1$ similar to those in EDJoin.

4.1.3. VChunk: The main diffusion in gram-based approaches appears in existence of the overlapping grams. For each string, we divide it into different disjoint substrings, or chunks, and we only need to process and index these substrings. This will give a passive improvement of space usage: for each string of length l, we only need to use 4l/avg_chunk_len bytes to store the hashed representation of the chunks rather than 4l bytes for gram-based methods. The signatures of VChunk are variable-length chunks, called "vchunk"[5].

Chunk boundary dictionary (CBD) is the key part of signature generation. It is a set of rules used to separate the strings into several chunks. For example, "pv" can be a rule. For each string, if it contains a substring "pv", VChunk splits the string just after the substring. For example, the string "pvlmod" will be split into two chunks: "pv" and "lmod" by this rule.

VChunk explains tail-restricted Chunk Boundary Dictionary which is a subset of all the CBDs and proves that if using a tail-restricted CBD to separate strings, any edit operation will prune at most 2 chunks. Thus the gram-based technique can be applied to the chunks cut by CBDs. Tail-restricted CBD classifies the character alphabet Σ in the strings into the two disjoint subsets: the prefix character set P and the suffix character set Q i.e., $P \cup Q = \Sigma$ and $P \cap Q = \varphi$. All the rules in the CBD can be given by regular expression [P]*[Q] which indicates strings with the arbitrary numbers of characters from P and a single character from Q. If this CBD is used to split string "pvlmo", it will get three chunks {"pvl", "m", "o"}. VChunk uses similar techniques of EDJoin for the similarity join using the CBD. In this algorithm, the most important thing to be considered is to select a proper method to btain the CBD.

CONCLUSION

After providing a comprehensive study of all the above mentioned algorithms, we can now say that-

1. Similarity join problem becomes hard for datasets having short strings, when working with edit distance. There are a large number of results and hence it is difficult to find efficient filtering techniques.
2. For the gram based methods, deciding on a appropriate value of parameter q is important, because it has effect on the performance. It is not easy to select a suitable parameter q in order to gain high performance. Generally q should be neither very small nor very large.

REFERENCES

[1] Website-https://en.wikipedia.org/wiki/String_metric.
[2 } R. J. Bayardo, Y. Ma, and R. Srikant: Scaling up all pairs similarity search.
[3] Chuan Xiao, Wei Wang and Xuemin Lin, EdJoin: An Efficient Algorithm for Similarity Joins With Edit Distance Constraints.
[4] J. Qin, W. Wang, Y. Lu, C. Xiao, and X. Lin : Efficient exact edit similarity query processing with the asymmetric signature scheme.

[5]W. Wang, J. Qin, C. Xiao, X. Lin, and H. T. Shen. Vchunkjoin: An efficient algorithm for edit similarity joins. IEEE Trans. Knowl. Data Eng., 25(8):1916–1929, 2013.
[6] 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 VLDB, pages 491–500, 2001.
[7] C. Li, J. Lu, and Y. Lu : Efficient merging and filtering algorithms for approximate string searches.