An Efficient Framework for String Similarity Continuous Query on Data Stream

Jia Cui, Lei Shi, Juan Li, Zhaohui Liu
China Information Technology Security Evaluation Center
Beijing, China

Abstract—**With rapid development of network technologies, the data accessing paradigm has been transferred from disk-oriented to "on-the-fly" data stream. The string similarity query on data stream has a broad prospect of application, especially in information security area and network monitoring. Due to the characteristics of stream and limitations of computing resources, the current methods based on static dataset cannot support stream efficiently. To solve these challenges, a framework named F2SCQ (framework of string similarity continuous query) based on filtering and verifying approach is pro-posed. It adopts basic window mechanism to update the sliding window, and the improved asymmetric signature (IAS) scheme to extract signature is proposed. Moreover two new filtering algorithms: Pre-Prune Filtering (PPF) and Count Filtering on Stream (CFS) are proposed. The experiments show that F2SCQ achieves high performance over high rates data stream. Compared to q-gram and asymmetric signature scheme, IAS achieves 50% and 20% faster extraction speed and 45% and 9% less storage overhead. The proposed filtering algorithm also achieves faster filtering speed and generates fewer candidates. F2SCQ minimizes the time and space complexity.**

Keywords:Continuous Query, Streaming Processing, Similarity Query

I. INTRODUCTION

Recently, the string (set) similarity query or join has be-come a leading paradigm in data processing area for the last decade, especially in cross-discipline subject. For instance, data integration and cleaning in database [2], the similar pattern matching of protein or DNA sequences in bioinformatics [3], the spell checking/correction in applications [4]. The mainstream researches in this area are based on static datasets. However the researches of keyword search on data stream are mainly based on exactly string matching.

The string similarity query on data stream has a broad prospect of application, especially in information content security and spam mail filtering etc. But the distinct differences between static dataset and data stream environment bring us great challenges. The characteristics of static dataset are finite, stable reading rate, patch up-date and store in the disk. The characteristics of data stream are boundless, fluctuant input rate, real-time update, processing in memory and do not store in disk. Those differences at least bring us three challenges: the first is on the index aspect. The boundless data feature makes it impossible to cache all data stream and make index. The creation and updating should be done in memory and all operations are restrained to the limited memory resource. Moreover, the sliding window model of data stream makes the data expired and forces the updating speed of index should be finished in real-time. The second is on real-time response aspect. Due to the fluctuant speed of data stream, the peak of input may occur that makes the query method should handle this situation efficiently and return the results in time as well. The third is on the difference of query mode. In the mode of query on static dataset, the query string is mutative and target is relatively fixed. However, in the mode of query on data stream, the query is relatively fixed and target is mutative.

There has been a great deal of literatures on similarity search/join on static dataset or massive data at present; unfortunately, there is not much research on string similarity query/join on data stream. Due to the characteristics of data stream and the limitations of computing re-sources in stream processing system, the current methods based on static dataset cannot handle the data stream environment efficiently. To solve the challenges mentioned above, our work aims to study a framework which minimizes the time and space complexity, and has a great real-time performance and peak value processing capacity. Hence a framework named F2SCQ (framework of string similarity continuous query) is proposed. It inherits the filtering and verifying approach and adopts the edit-distance as the string similarity metric. To solve sliding window index updating problem, we adopt basic window mechanism to support updating. We proposed an improved asymmetric signature scheme (IAS) to ex-tract string signatures on data stream and query strings. We also proposed two filtering algorithms named Pre-prune filtering (PPF) and count filtering over streams (CFS). The former is used to minimize the index overhead; the latter reduces execution time and generates fewer candidates that need to be verified.

The rest of this paper is organized as follows. Section 2 reviews related works on the areas of string similarity and data stream. Section 3 introduces the preliminaries. Section 4 presents the formal definition, sliding window model, IAS scheme, filtering & verifying algorithms in F2CSQ. Section 5 presents the experimental results and performance evaluation. Section 6 summarizes and concludes our work.

II. RELATED WORKS

In this section, we review the current research status, concerning string similarity area, data stream area and keyword search area. The string similarity problem has been proposed

978-1-7281-0120-0/18/$31.00 ©2018 IEEE

for several decades. It aims to find the approximate matching string with a similarity threshold occurred in the static collection of documents, see G. Navarro [1] for an excellent survey. The research field of string/set similarity includes: string similarity query and string (set) similarity join. The formal definition of string similarity query problem is defined as follow:

Gives a query Q, a similarity function $sim(-,-)$ (or distance function $dist(-,-)$) and a similarity (or distance) threshold τ, the returned results r should satisfy that $sim(r,Q)\geq\tau$ ($dist(r,Q)\leqslant\tau$).

At present two different approaches are proposed to solve this problem which are filtering & verifying based (i.e., signature based) approach and the Trie based approach. The basic idea of filtering & verifying approach is relaxing the edit-distance metric to the overlap of the signature set. It has two phases: in the first phase, a signature extraction scheme is adopted to obtain the strings signatures in the dataset and create an inverted index ID based on them. When the query Q is submitted, the same signature extraction scheme is applied and obtains the signature set G_Q. The following steps use filtering algorithms to compute the lower bound $L_{,Br;s}$ of overlap according to edit-distance threshold τ and filtering out the survived strings as candidates. In the second phase, real edit-distance function is applied to compute the final result. Note that the performance of signature extraction scheme and filtering algorithm are crucial in this approach. The current proposed string signature extraction schemes are:

1). Fixed-length q-gram [9]. The q-gram is an ordered pair (g,i), in which g is the substring that starts from i-th position in s with the length q.

2).Fixed-length q-chunk. The q-chunk is an ordered pair (c,j), in which c is the non-overlap and adjacent substring that starts from j-th position in s with the length q.

3).Variable-length gram. VGRAM [5] is a variable-length gram extraction scheme. It extracts all k-gram ($q_{\min} \leq k \leq q_{\max}$) and constitute them as a frequency Trie. The gram with higher frequency is selected to generate a dictionary as the signature extraction criterions.

4).Variable-length chunk. VCHUNK [16] is a variable-length chunk extraction scheme. It counts all character frequency in dataset and selects the characters with higher frequency as chunk boundary dictionary (CBD). Tail-restrict strategy based on CBD is applied to generate variable-length chunk of strings in dataset.

5).Partition-based signature [13]. Given an edit-distance threshold τ, it partitions strings into τ+1 partitions. Accord to the pigeonhole principle, there should be at least one overlap in the partition of string r and s, if they satisfy $ed(r,s) \leqslant\tau$.

6).Asymmetric signature [10]. It adopts two different signature extraction schemes to dataset and query, which are q-gram and q-chunk.

The popular filtering algorithms proposed at present are: length-filtering [15], count-filtering [15], position-filtering [15], prefix-filtering [2], content-based mismatch filtering [9], and location based mismatch filtering [9] etc.

The second approach is based on Trie. J. Wang etc. [11] proposed a method to solve the problem of similarity join/search on dataset. Its basic idea is to build a Trie index on the string set and then traverse it in pre-order from the root node to leaf node according to the query string. The active-node set A_n is computed while visiting node n. It iteratively computes all active-node sets on the path until reaching the leaf node. The leaf nodes in A_n is the similar string that satisfy the threshold τ . Although this approach avoids the verifying phase, it has highly time and space overhead in generating Trie index, and Trie is hard to update. So we do not consider this approach in our work.

In research area of data stream, an overwhelming number of real or prototype data stream management systems (DSMS) have been proposed. The Stanford's Stream Data Management (STREAM) project [17] is a general purpose data stream management system that addresses problem ranging from basic theoretical results to implementing a comprehensive prototype. The TelegraphCQ project [18] is focus on meeting the challenges that arise in handling large number of CQs over high-volume, highly-variable data streams. The Aurora project [19] is building a new data processing system targeted exclusively towards stream monitoring applications. The UTA's MavStream project [20] is designed to address stream processing holistically from a QoS perspective.

In the research area of keyword search on stream, V. Hristidis etc. [7-8] are the first to propose the studies of processing continuous queries on stream. [21] inherits the idea from DISCOVER for executing a keyword searching over relational data streams, it generates CNs for data streams and maps operator trees of CNs into operator mesh to share buffers, and it also returns MTJNTs as results. L.Qin et al. [22] improved the method of [23] by constructing a data structure called L-Lattice to share query processing cost among all CNs. Golab et al. All these works are based on exactly matching.

We are inspired by the current situation that is the existing proposed keyword search approach on data stream are based on exactly match, which largely restrains the effectiveness and flexibility of stream query system. Considering these situations, if the stream data items come from network interface, some error-bits may exist in the packets; if the data items come from applications that need users to type in, some misspelled strings may exist. The current system could not handle these queries effectively. For example, suppose a DSMS received several records with fixed schema as shown in Table 1. If a user registers a query with keywords "Marios Hadijeleftherious" and the system will return nothing, because the string "Hadijeleftherious" and "Hadjieleftheriou" are not exactly match. However, if the DSMS is integrated with string similarity query feature, record 3 will be returned as a result.

TABLE I. THE TABLE OF RECEIVED ITEMS

Record ID	*String*	*Time Stamp*
1	*Gonzalo Navarro*	*346332231*
2	*Vagelis Hristidis*	*347823432*
3	*Marios Hadjieleftheriou*	*398762343*
4	*Younghoon Kim*	*427221223*
5	*Gonzalo Navaro*	*462678743*

III. PRELIMINARIES

A. String Signatures and Inverted Index

Let Σ be an alphabet. A string s is composed of the characters in Σ and $s \in \Sigma^*$, we use "$|s|$" to denote the length of string s. Given a string s and a positive integer q, a positional q-gram is a pair (g, i), where g is the q-gram signature which is a substring start at i-th character with length of q. The set of positional q-grams of s, denoted by $G_g(s,q)$, is obtained by sliding a window with length q over the character of string s. The size of $G_g(s,q)$ is denoted by $|G_g(s,q)|$. Apparently $|G_g(s,q)| = |s| - q + 1$. For example, assuming a s = "alphabet" $q = 2$, thus $G_g(s,q)$ = {(al,1), (lp,2), (ph,3), (ha,4), (ab,5), (be,6), (et,7)} and |G(s,q)|= 7.

The inverted index is a term-oriented mechanism for quickly searching documents containing a given term. It consists of two major components: terms and posting lists. The posting list related to a specific term contains the information about the occurrences of term. The q-gram inverted index uses the q-gram as indexing terms.

B. Similarity and Dissimilarity Functions

There are two types of functions to measure the similarity between strings: similarity functions (e.g. Jaccard similarity) and dissimilarity functions (e.g. edit distance). If two strings r and s are similar, their similarity is no smaller than a given threshold φ , or their dissimilarity is no larger than a given threshold τ . The threshold φ for similarity function belongs to [0, 1] and the threshold τ for dissimilarity function is a non-negative integer. Typically, the edit-distance (a.k.a Levenshtein distance) between r and s is the minimum numbers of single character edit operation that is needed to transform between r and s. The operations include insertion, deletion and substitution. The classical algorithm of computing edit-distance is dynamic programming. Its time complexity is $O(n^2)$. We denote $ed(r,s)$ as edit-distance function between r and s such that $0 \le ed(r,s) \le \max(|r|,|s|)$. For example, r="surgery" and s="survey", then $ed(r,s)$=2.

C. Continuous Query with Sliding Window

First of all, we give some definition related to data stream in this paper.

Definition 1 (Data Stream). A data stream S is composed of a series of tuples, the form of a tuple is <s,ts>, thus

$$S = \{<s_1,ts_1>, <s_2,ts_2>, \ldots,<s_i,ts_i>,\ldots>$$

in which s_i denotes the content of tuple s_i, ts_i denotes the timestamp of it. The timestamp could be assigned by the generating source or by DSMS system when receiving it.

The sliding window is an extensively used model in data stream area. It is defined as a historical snapshot of a finite portion of stream at any time instant. This implies that the old data items are discounted and less valuable than the recent ones.

Definition 2 (Sliding Window). DSMS maintains a buffer window with a constant time interval T. With the newly arrived tuples filled in the buffer, the old tuples expired according to their timestamps. As the time passed, this buffer can be treated as sliding. We denote SWS[t-T,t] as the sliding window at the moment of t.

Traditional continuous query with sliding window was defined as processing repeatedly against each newly arrived record to produce results. Obviously it is expensive and impractical to re-evaluate the entire sliding window against each newly arrived record, especially when the arrival rate is very high.

IV. F2SCQ

An intuitive approach of processing the string similarity query on data stream is to compute the real edit-distance with each newly arrived data against the query and compare to the threshold. But this approach is limited to the highly time complexity due to the edit-distance dynamic programming algorithm. In this section, we will discuss the F2CSQ framework. Section 4.1 discusses the sliding window model and overview of the framework. Section 4.2 presents the improved asymmetric signature extraction scheme. Section 4.3 discusses filtering algorithm. Section 4.4 discusses the verifying algorithm.

A. Overview

In this paper, we only consider the single stream situation. Assuming that there is a data stream management system receives an incoming data stream S. Some definitions related to this work are as follow.Based on definition 2, on the stream with highly input rate, the updating of the sliding window is highly frequent. So it is hard to support the string similarity query based on the sliding window under this circumstance. So we introduce the basic window.

Definition 3 (Basic Window). Basic window is the sub-window of sliding window which is non-overlap and adjoin with each other. It inherits the attributes of sliding window and a sliding window denotes as

$$SW = \bigcup BW_i$$

in which BWi means the i-th basic window in the sliding window. Note that every basic window preserves the same time-interval or number of tuples.

According to definition 3, the sliding window model adopted in F2SCQ is composed by a set of basic window, that is, the system maintain a queue of basic window and a queue of basic window index separately. When a new tuple arrives, first it is filled to the basic window directly. If the basic window is full, the basic window signature index is generated, meanwhile the basic window and its index will append to their corresponding queues. Figure 1 shows the basic window based sliding window model.

Note that each basic window also has an attribute of timestamp, which is the max value of the timestamp in the tuples belongs to the basic window. All operations over the sliding window index are based on basic window. Different from [6], the purpose of introducing the basic window in our work is to partition the sliding window into smaller piece. Thus

when facing the stream with highly input stream, this mechanism could not only support the updating of sliding window index, but also lower down the frequency of updating and triggering query. Unfortunately, it may bring the errors in result due to the size of basic window, i.e., updating granularity. Based on the sliding window model defined above, we give a formal definition of string similarity continuous query on stream.

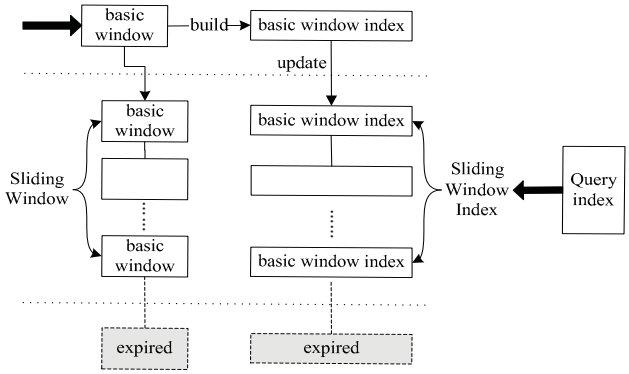

Figure 1. The sliding window and index model of F2SCQ

Definition 4 The query registered to the data stream system is denoted as follow quintuple form:

$$Q = \{K, \tau, q, T, t\}$$

in which K is the set of query string, K={k_1, k_2, …, k_n},τ is the edit-distance threshold, q is the signature length, T is the time interval of sliding window, t is the time interval of basic window. The sliding window contains $|T/t|$ windows. At the time constant t_k, when the sliding window is updated by a basic window, the query is triggered against the entire sliding window and returns the results as follow.

$\{s \in SW_S\ [t_k\text{-}T, t_k]$

$= \bigcup BW_i \mid BW_i.ts \in [t_k\text{-}T, t_k]$ and $ed(s,k_j) \le \tau\ \}$

Note that in this paper, the data stream management system never stores the data stream on hard-storage and it processes the query in memory.

B. Signature Extraction Scheme

The string signature extraction scheme will directly affect the speed of index creation and updating, so it is the crux in determining the real-time query performance on stream. In this work, we improved the asymmetric signature extraction scheme in [10] (IAS for short) to apply in the string similarity query processing on stream. The scheme description is defined as follow.

Extract *q*-gram signatures on query string: First, we fill q-1 special characters "$" at the end of query string r, r'= r+|q-1|"$", to make sure that every single character in r has a q-gram signature. Then we extract q-gram in r'.

Extract *q*-chunk signatures on stream. The original asymmetric signature extract scheme fills (q-|s|mod q) special characters "$" at the end of string s, then extract the q-chunk signatures. We notice that when (|s| mod q) = 0, it will generate the signature with length q all filled with "$". But this signature will not match any q-gram from r'. It is redundant so we improved it as follow (q-chunk'). First we fill δ special characters at the end of s, s'=s+ δ "$", then extract q-chunk signature on string s of stream.

$$\delta = \begin{cases} q - |s| \bmod q, & (|s| \bmod q) \neq 0 \\ 0, & (|s| \bmod q) = 0 \end{cases}$$

F2SCQ Framework

Input: Stream S, query Q={K, τ , q, T, t}
1. I_K=*CreateQueryIndex*(K, q)
2. **While** (a new tuple arrives $s \in S$){
3. **If** (current BW_i is not full)
4. *Fill s to* BW_i
5. **Else** { // BW_i is full
   i. I_{BW}=*CreateBWIndex*(BW_i, q)
   ii. $BW_i.ts$ = *max*(ts_i),
   iii. *Append* BW_i and I_{BWi} to the end of queue
   iv. Create a new basic window BW_{i+1} and fill s in it.
   v. Trigger the ***Query_Thread*** }}

End Algorithm

Query_Thread
1 Remove the expired basic window, get the current snapshot of sliding window index I_{SW};
2 Filter phase : C=***Filtering_Step***(I_{SW}, I_K)
3 Verify phase: R=***Verify_Step***(C, K, τ)

Figure 2. The pseudo-code of F2SCQ

CreateQueryIndex (K, q)
Input: set of query strings K, length of signature q;
Output: Query Index I_K
1.Initialize inverted index $I_K \leftarrow \Phi$
2.**For** each keyword k_i in K
3. ki'←pad q-1 "$" at the end of k_i
4. **For** (int j=1 ; $j \leqslant |k_i'|$-q+1 ; j++)
5. Add $k_i'[j,j+q]$ and <i, j> to I_K
6.return I_K
End Algorithm

Figure 3. The pseudo-code of *CreateQueryIndex*

The two different signature extraction schemes (q-gram and q-chunk') above compose our improved asymmetric signature scheme. Note that the creation and update of index on stream is all based on basic window, and organized as inverted index.

C. Filtering Algorithm

In our F2SCQ framework, another crucial part that affects the real-time performance of the queries on stream is the filtering algorithm. We argue that the qualified filtering algorithm on stream should satisfy at least two points: fast filtering speed and strong filtering power. Obviously, the first point will affect the processing time. The second point will affect the size of candidates which need to be verified in verifying phase. Different from the methods based on the static dataset, we cannot get all data to build an index over it. So we should work on the filtering algorithm carefully to make sure

the filtering phase to be more powerful. In our work, first we make some experiments to test if any existed algorithm is fit for the stream environment. Then we propose more two filtering algorithms the strength the filtering phases.

CreateBWIndex (s, I_{BW}, q)

Input: tuple $s \in S$, length of signature q;
Output: Index of basic window I_{BW}

```
1.  When s ∈ S arrived
2.   If(BW_K is full) {
3.       For each tuple s_i in BW_k {
4.           If |s_i| mod q = 0   δ ←0;
5.           Else  δ ←q - |s_i| mod q;
6.           s_i'←pad δ "$" at the end of s_i ;
7.           For(int j=1; j≤|s_i'|/q; j++)
8.               Add s_i'[j*q, (j+1)*q] and <i, j*q> to I_BW}
9.       Append I_BW and BW_K to the corresponding queue}
10.  Else(BW_k is full) {
11.      Create a new BasicWindow BW_{k+1};
12.      Append s to BW_k; }
```

Figure 4. The pseudo-code of CreateBWIndex

The existed filtering algorithms can be used in F2SCQ framework are:

Length Filtering. Based on the definition of edit-distance, if the string s on stream satisfies the edit-distance threshold with any query string k_i, s must satisfy the following condition.

$$Min(|k_i|) - \tau \le |s| \le Max(|k_i|) + \tau$$

Position Filtering. The matching pair between q-chunk signature from string on stream and q-gram signature from query, not only their content should be the same, but also their position should within the τ .

Through the experiments, we found that the popular Prefix Filtering algorithm is not fit for the stream environment. There are three reasons. First, the premise of prefix filtering is the global ordering O, which is usually decided by the idf (inversed document frequency) of signatures. The prefix of signatures are extracted according to O. Unfortunately, in stream environment, due to its boundless and mutative features, it is hard to determine the global ordering. Second, if we obtain the global ordering just by scan the signatures generated by the query strings, it makes no sense since the idf of these signatures may be equal. Third, even if the global ordering has been determined, it is also time consuming to sort these signatures. For the three reasons above we do not consider Prefix Filtering in our framework.

In addition, we also propose two new filtering algorithms that fit for stream environment: Pre-prune Filtering and Count-Filtering on Stream.

Pre-prune Filtering (PPF). This filtering algorithm is designed for the basic window index creation phase. Its basic idea is that since the query string on data stream is relatively fixed, we can think target of creating the basic window index is clear, i.e., some filtering can be done before the index creation to reduce the memory cost. First, we retrieve the tokens from q-gram index by preprocessing the query strings. We treat these tokens as the initial value of each basic window index. It is intuitively to know that one edit error only affect one q-chunk. So when generating the q-chunk signatures, we do error counting. If the number of mismatched q-chunk against the initial value of has exceeded τ , this string can be pruned, else these signatures are added to IBW. Figure 5 show the description of it.

Pre-Prune Filtering Algorithm

Input： index of query string I_K, basic window BW, length of signature q, edit-distance threshold τ ；

```
1.  Retrieve the tokens from I_K as the initial value of I_BW;
2.  For (each s_i in BW){
3.    For ( int j=1; j≤|s_i|/q; j++){
4.      Get signatures set A^{si} = s_i'[j×q, (j+1)×q])}
5.      For (each tokens in A^{si}){
6.        Compare to tokens in I_BW , N=mismatched number;  }
7.        If (N≥τ) continue;  //go to next string in BW
8.        Else (add A^{si} to I_BW);  }
End Algorithm
```

Figure 5. The pseudo-code of Pre-Prune Filtering

Count Filtering on Stream (CFS). [10] proposed an count filtering **algorithm** based on asymmetric signature scheme. Its lower bound equation of overlap is as follow.

$$|G_g(r,q) \cap G_c(s,q)| \ge \lceil |s|/q \rceil - \tau \qquad (1)$$

Equation (1) illustrates that when the length of signature and edit-distance threshold have been determined, the lower bound of overlap is decided by the length of the string which q-chunk extraction works on. If we use this filtering directly to our F2CSQ framework, it has one drawback, that is, we need to access the length of the strings in basic window when computing the lower bound, since we do the computing over the index, or we need to ***maintain*** an additional index for the string length. It brings us more cost of the access and index. So we propose the CFS filtering algorithm to address this problem. Its basic idea is as follow.

Assuming that two string r and s that satisfy $ed(r,s) \le \tau$. According to Length Filtering, they must satisfy the equation $||r|-|s|| \le \tau$. Substituting it into Eq. (1), we obtain Eq. (2) as follows.

$$\left\lceil \frac{|r|-\tau}{q} \right\rceil - \tau \le \lceil |s|/q \rceil - \tau \le \left\lceil \frac{|r|+\tau}{q} \right\rceil - \tau \qquad (2)$$

The meaning of value on both sides in Equation (2) is the range of the overlap lower bound, i.e., the number of overlapping signatures between the strings within the length subtraction of τ . When the value of right side minuses the left side is no larger than 1, the overlap can be determined. When it is larger than 1, it is more likely that the real overlap is larger than the value of left side. So we could tighten the lower bound on the left side, we obtain Equation (3) as follows.

The complexity analysis for CFS filtering Algorithm: in data stream environment, if we adopt Equation (1) to compute the lower bound of overlap, it is necessary to compute the

lower bound for each string in stream. Its time complexity is O(n), in which n is the number of strings in stream. If we adopt Eq. (3) , the lower bound is decided by the query string and reduce the time complexity to O(1) since we just need to compute the lower bound once. It accelerates the execution speed and improves the real-time performance. However, Eq.(3) may lose some accuracy of the results. See the experiments results in section 5.3

$$G_g(r,q) \cap G_c(s,q) \begin{cases} \geq \left\lceil \frac{|r|-\tau}{q} \right\rceil - \tau, & 2\tau < q \\ > \left\lceil \frac{|r|-\tau}{q} \right\rceil - \tau, & 2\tau \geq q \end{cases} \quad (3)$$

D. *Verifying Algorithm*

Based on the observation of prefix-pruning in Trie [11], we understand that it does not need to compute entire matrix to finish the verification. Hence, this observation inspired us to implement it as our edit-distance verifying algorithm. The basic idea is as follow. Suppose given two strings $r=r_1r_2...r_n$ and $s=s_1s_2...s_m$. Let M denote a matrix with n+1 rows and m+1 columns while computing the edit-distance. $M[i,j]$ denotes the edit-distance between the prefix string of r[1:i] and s[1:j]. Assume the algorithm is computing k-th row, take $M[k,k]$ as principal diagonal, we just need to compute and verify the content of cells in $M[k][k-\tau, k+\tau]$. If all cells in this range are larger than τ , which means the string does not satisfy the $ed(r,s) \leq \tau$. So we could do early-termination. Figure 6 is the implementation of the algorithm.

V. EXPERIMENTS

We implement a prototype system based on F2SCQ to evaluate its feasibility and performance. The operating system is Linux CentOS 6.3, the kernel version is 2.6.32, java version is 1.6.0. We have two computers; each of them has dual-core CPU (2100MHz), 4GB memory. One computer worked as a client, reads the static dataset and sends them as UDP packets. Another computer is working as server, receives UDP packets and processes the continuous queries.

The experiment is made on the open-dataset: English-Dict. What we concern the most is the performance of IAS scheme, including the time and space cost; the performance of our proposed filtering algorithm, and the system peak value processing capacity.

A. *IAS Performance*

In this section, we evaluate the performance of q-chunk' in IAS string signature extraction scheme, compared to the traditional q-gram, the q-chunk original asymmetric signature scheme. We compare them in the time and space cost for building the basic window inverted index. As Fig.7 and Fig.8 show, the x-coordinate means the number of tuples that the basic window keeps, the Y-coordinate means the time cost and memory cost separately.

		s	i	m	i	l	a	r
	0	1	2	3	4	5	6	7
s	1	0	1	2	3	4	5	6
a	2	1	1	2	3	4	4	5
m	3	2	2	1	2	3	4	5
a	4	3	3	2	2	3	3	4
l	5	4	4	3	3	2	3	4
a	6	5	5	4	4	3	2	3
r	7	6	6	5	5	4	3	2

Figure 6. The Matrix of Computing Edit-distance

Early-Termination Algorithm

Input: string r and s, edit-distance threshold τ ;

1. Initial the matrix M,
   For(int i = 0 ; i < |s| ; i++) $M[0][i] = i$;
   For (int j = 0 ; j < |r| ; j++) $M[j][0] = j$;
2. **For** (int i = 1 ; i < |s| ; i++){
3. **For** (int j = 1 ; j <|r| ; j++){
4. **If** ($s[i]$= $r[j]$)
   $M[i][j] = M[i-1][j-1]$;
5. **Else** $M[i][j]=min(M[i-1][j-1],M[i][j-1]+1,M[i-1][j])+1$;}
6. **If (All** $M[i][i-\tau : i+\tau] \geq \tau$)
   Return False;
7. **Else** continue; }
8. **If** ($M[|s|][|r|] > \tau$) **Return** False;
9. **Else Return** True;

Figure 7. The implementation of Early-Termination algorithm

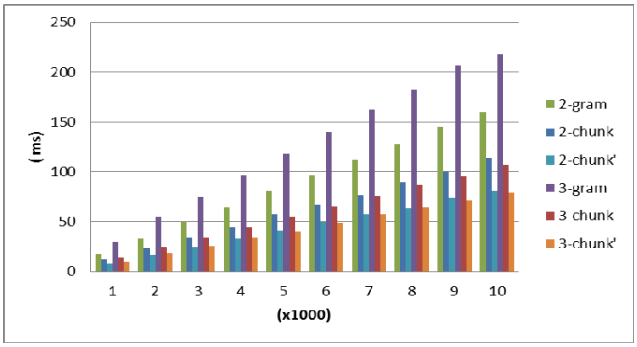

Figure 8. Comparision of basic window index generation time

In the index creation time aspect, the q-chunk' achieves the best performance. Compare to q-gram, it reduce 50% percent of time cost. Compare to q-chunk, it reduce 20% percent of time cost. In the index memory cost aspect, Compare to q-gram, it reduce 45% of memeory cost. Compare to q-chunk, it reduce 9% of the memory cost.

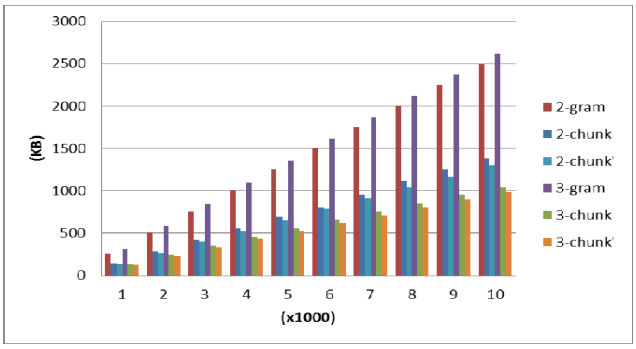

Figure 9. Comparision of basic window index size

Conclusion 1. The q-chunk' in IAS is excellent for data stream environment in both time and memory cost.

B. Filtering Algorithm Evaluation

In this section, we evaluate the execution time and filter power of proposed CFS algorithm compared to the exsited count-filter algorithm on stream. Suppose each basic windows is composed of 1000 tuples. Fig.10 shows the sampling execution time.

We also did three query test on this stream. The query string we picked is "*happy*", "*hello*", and "*instrumentation*". we chose q with the value 2 and 3, edit-distance threshold with the value of 1 and 2. Fig.11-14 shows the sum of the candidate size on processing the stream.

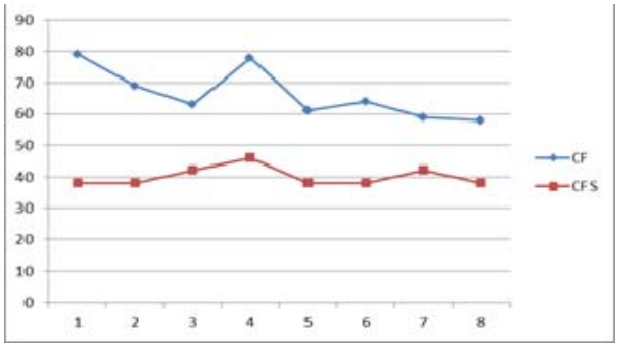

Figure 10. Comparision of execution time of CF and CFS

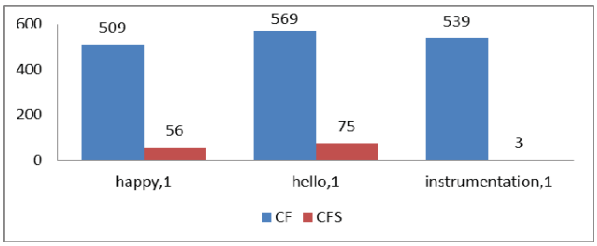

Figure 11. Comparision of candidates size (τ =1, q=2)

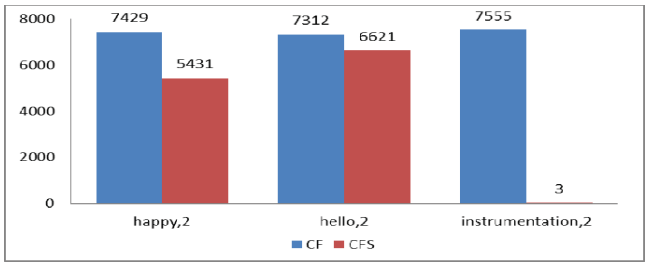

Figure 12. Comparision of candidates size (τ =2, q=2)

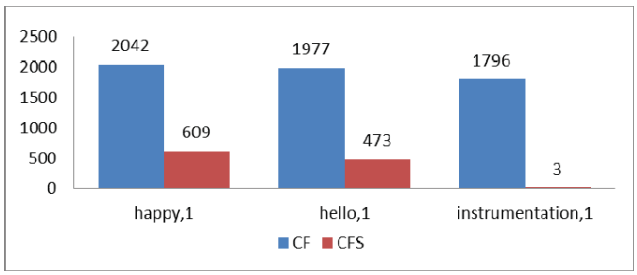

Figure 13. Comparision of candidates size (τ =1, q=3)

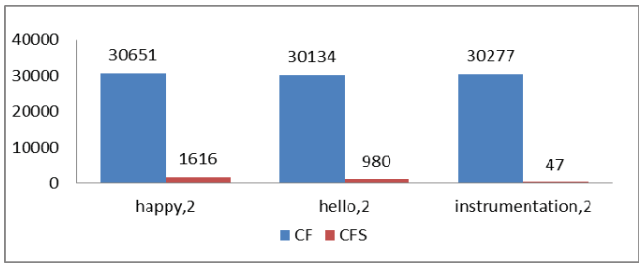

Figure 14. Comparision of candidates size when (τ =2, q=3)

TABLE II. **COMPARISON OF RESULT ACCURACY**

	q=2		q=3		
	CF	CFS	CF	CFS	Result
happy, τ =1	7	7	7	7	7
happy, τ =2	60	60	60	***36***	60
hello, τ =1	9	9	9	9	9
hello, τ =2	110	110	110	***40***	110
instrumentation, τ =1	2	2	2	2	2
instrumentation, τ =2	3	3	3	3	3

Conclusion 2. CFS algorithm acheives faster processing speed than the original count filter algorithm,and it generates fewer candidates that need to be verified. When q and τ is bigger, the CFS will lose some accuracy as it tightens the lower bound.

C. Peak Value Capacity

In this section, we evaluated the peak value processing capacity in three methods: 1. directly using edit-distance function to support query. 2. In F2SCQ sliding window model,

using existed methods based **on** static dataset. 3. In F2SCQ sliding window model, using IAS, PPF and CFS. Table 2 shows the results. Note all these methods using Early-Termination algorithm.

TABLE III. **COMPARISON OF PEAK** VALUE **TEST RESULTS**

	Time(/tuple)	Peak value
ED-ONLY	4.1us	3.2×10^5
F2SCQ sliding window + existed method	1.3us	7.9×10^5
F2SCQ(IAS+PPF+CFS)	0.8us	1.2×10^6

Conclusion 3： The basic window based sliding window model integrated with original asymmetric signature scheme and related filtering can processing a certain speed of stream. Compare to naïve method of computing the edit-distance directly, it promotes twice peak value processing capacity. Our F2SCQ framework promotes 4 times of it, and it promotes about 43% percent of it.

VI. CONCLUSIONS

With the rapid development of network technology, the real-time query with similarity feature has a broader prospect of application. In this paper, we proposed a framework to address the problem of string similarity continuous query on data stream. It aims to minimize the time and space complexity, and obtain a higher peak value processing capacity. The experiments show that our framework handles the query efficiently. Our future work is to study the problem of on-line self/multiple sliding window similarity join.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (No. U1536118).

REFERENCES

[1] G. Navarro, "A guided tour to approximate string matching", ACM computing surveys (CSUR), vol 33, no 1, pp 31–88, 2001.

[2] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning", 2006, pp 5–5.

[3] H. E. Williams and J. Zobel, "Indexing and retrieval for genomic databases", in IEEE Transactions on Knowledge and Data Engineering, 1998.

[4] S. Chaudhuri and R. Kaushik, "Extending autocompletion to tolerate errors", in Proceedings of the 35th SIGMOD international conference on Management of data, 2009, pp 707–718.

[5] C. Li, B. Wang, and X. Yang, "VGRAM: improving performance of approximate queries on string collections using variable-length grams", 2007, pp 303–314.

[6] L. Golab, S. Garg, and M. Özsu, "On indexing sliding windows over online data streams", Advances in Database Technology-EDBT 2004, pp 547–548, 2004.

[7] V. Hristidis, O. Valdivia, M. Vlachos, and P. S. Yu, "A System for Keyword Search on Textual Streams", in Proceedings of the Seventh SIAM International Conference on Data Mining, Minneapolis, Minnesota, USA, 2007.

[8] V. Hristidis, O. Valdivia, M. Vlachos, and P. S. Yu, "Continuous keyword search on multiple text streams", in Proceedings of the 15th ACM international conference on Information and knowledge management, 2006, pp 802–803.

[9] C. Xiao, W. Wang, and X. Lin, "Ed-Join: an efficient algorithm for similarity joins with edit distance thresholds", Proceedings of the VLDB Endowment, vol 1, no 1, pp 933–944, 2008.

[10] J. Qin, W. Wang, Y. Lu, C. Xiao, and X. Lin, "Efficient exact edit similarity query processing with the asymmetric signature scheme", 2011, pp 1033–1044.

[11] J. Wang, J. Feng, and G. Li, "Trie-join: Efficient trie-based string similarity joins with edit-distance thresholds", Proceedings of the VLDB Endowment, vol 3, no 1–2, pp 1219–1230, 2010.

[12] R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling up all pairs similarity search", In: Proceedings of the 16th international conference on World Wide Web, Banff, Alberta, Canada, 2007, pp 131–140. 2007.

[13] G. Li, D. Deng, J. Wang, and J. Feng, "Pass-join: A partition-based method for similarity joins", Proceedings of the VLDB Endowment, vol 5, no 3, pp 253–264, 2011.

[14] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In: Proceedings of the 32nd international conference on very large data bases, Seoul, South Korea, 2006.918–929

[15] 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: Proceedings of the international conference on very large data bases, Rome, Italy, 2001. 491–500

[16] W. Wang, J. Qin, X. Chuan, X. Lin, et al. VChunkJoin: An Efficient Algorithm for Edit Similarity Joins. IEEE Transactions on Knowledge and Data Engineering, 2012, 6(1):1

[17] STREAM: Stanford Stream Data Management (STREAM) Project. http://infolab.stanford.edu/stream/.

[18] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, "TelegraphCQ: continuous dataflow processing", in Proceedings of the 2003 ACM SIGMOD international conference on Management of data, 2003, p. 668–668.

[19] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management", The VLDB Journal, vol. 12, no. 2, pp. 120–139, 2003.

[20] Q. Jiang and S. Chakravarthy, "Data stream management system for mavhome", in Proceedings of the 2004 ACM symposium on Applied computing, 2004, pp. 654–655.

[21] A. Markowetz, Y. Yang, and D. Papadias, "Keyword search on relational data streams", in Proceedings of the 2007 ACM SIGMOD international conference on Management of data, 2007, pp. 605–616.

[22] L. Qin, J. X. Yu, and L. Chang, "Scalable keyword search on large data streams", The VLDB Journal, vol. 20, no. 1, pp. 35–57, 2011.

[23] A. Markowetz, Y. Yang, and D. Papadias, "Keyword search on relational data streams", in Proceedings of the 2007 ACM SIGMOD international conference on Management of data, 2007, pp. 605–616.