Available online at www.sciencedirect.com

ScienceDirect

AASRI Procedia 4 (2013) 313 – 318

AASRI Procedia

www.elsevier.com/locate/procedia

2013 AASRI Conference on Intelligent Systems and Control

# Constraint Based Sequential Pattern Mining in Time Series Databases - A two Way Approach

Vangipuram Radhakrishna[a*],Chintakindi Srinivas[b],Dr.C.V.Guru Rao[c]

[a]*Department of Information Technology, VNR Vignana Jyothi Institute of Engineering and Technology, Hyderabad, INDIA*
[b]*Associate Professor , Department of Computer Science and Engineering, Kakatiya Institute of Technology and Science, Warangal, INDIA*
[c]*Professor & Head, Department of Computer Science and Engineering, S. R Engineering College,Hasanparthy, Warangal, INDIA*

**Abstract**

Most of the patter search algorithms in literature are mostly string based and do not concentrate on finding sequential patterns with constraints in a given database. Also in query languages such as SQL or MySQL, the select clause does not allow the use of the non-aggregate functions as part of query compilation. The objective of this work is to propose a pattern mining algorithm which may be embedded into SQL or MySQL Query languages so that we can search for presence of sequential pattern in the database. The algorithm makes use of sliding sequential patterns with three look-ahead elements in the event of any mismatch.



***Keywords :*** sequence pattern;look ahead; time series;sequence tuple

## 1. Introduction

In this paper the idea is to first perform preprocessing of the sequential pattern before the search process. Instead of searching for a string based pattern, we perform search for a sequence of tuple or record values with user defined constraints. The elements of pattern considered are numerical type.

Most of the algorithms designed in literature are string based and are not developed by considering sequential patterns with constraints. An attempt is made in this work towards this direction by considering a time series database. We search for a sequential pattern of interest in a given database with each pattern element imposed with a user defined constraint.

The algorithm developed may be embedded in to any existing query languages so that the use of non aggregate functions is possible which makes the complexity of the query get reduced in terms of query processing and also the time efficiency. Section III, IV describes the proposed algorithm with a working example. The objective of this algorithm is to make the pattern shift by a length more than the pattern length as compared to the existing algorithms. The efficiency can be seen from the moves made by the pattern itself.

* Corresponding author. Tel.: +9700684242
*E-mail address:* radhakrishna_v@vnrvjiet.in


doi:10.1016/j.aasri.2013.10.046

## 2. Related Works

In [1] Wang and Kobayashi propose a pattern search algorithm for network security applications that computes a function called next before the start of the search process. The value defined by next is used in the event of a mismatch between the pattern and the text. In [8] the authors design a pattern search algorithm using two sliding windows. In [6] algorithm uses the concept of implications to find the relation between the pattern elements and is designed to find the sequence pattern in a given database. In [9] algorithm designed performs the two way pattern search using sliding patterns. The algorithm in [10] uses three sliding windows.

## 3. Proposed Work

The algorithm consists of two steps.
a) Pre-process sequence pattern with constraints.
b) Search for sequence pattern in the given database.

### *3.1 Preprocessing phase*

#### *3.1.1 Left Shift failure function*

$$\text{left\_shift}[L_1, L2, L3] = \text{minimum} \begin{cases} 1 & ; \ \text{pattern[length (pattern) -1]}=L_1 \\ 2 & ; \ \text{pattern[length (pattern) -1]pattern[length (pattern) -2]}=L_2L_1 \\ \text{length (pattern) -index} & ; \ \text{pattern[index]pattern [index+1] pattern [index+2]} =L_1L_2L_3 \\ \text{length (pattern) +1} & ; \ \text{pattern [0] pattern [1]} =L_2L_3 \\ \text{length (pattern) +2} & ; \ \text{pattern [0]} =L_3 \\ \text{length (pattern) +3} & ; \ \text{else} \end{cases} \quad (1)$$

#### *3.1.2 Right Shift failure function*

$$\text{Right\_shift}[L_1, L2, L3] = \text{minimum} \begin{cases} \text{length (pattern) +2} & ; \ \text{pattern [length (pattern) -1]} = L_1 \\ \text{length (pattern) +1} & ; \ \text{pattern [length (pattern) -1] pattern [length (pattern) -2]} = L_2L_1 \\ \text{index +3} & ; \ \text{pattern [index] pattern [index+1] pattern [index+2]} = L_1L_2L_3 \\ 1 & ; \ \text{pattern [0] pattern [1]} =L_2L_3 \\ 2 & ; \ \text{pattern [0]} =L_3 \\ \text{length (pattern) +3} & ; \ \text{else} \end{cases} \quad (2)$$

In the preprocessing phase failure functions are first computed. In case of any mismatch in the search process the algorithm uses three attribute values of the input present immediately after the aligned sliding pattern. These attribute values are called look ahead elements.

The algorithm for the preprocessing of the sequence pattern is given in fig 1 below.

```
Begin algorithm
left_shift    ← length (pattern) +3;
right_shift   ← length (pattern) +3;
for each pattern element Pattern [index] from i ← 0 to length (pattern) -3 do
     Compute Next [index] as Next [index]  ← length (pattern) –index;
                                 next [index]  ← index+2
end for
      If (pattern [length (pattern) -1] = =L1) then
                {left_shift← 1; right_shift← length (pattern) +2}
else If (pattern [length (pattern) -1] pattern [length (pattern) -2] = =L2L3) then
                {left_shift←2; right_shift←length (pattern) +1}
else If (pattern [index] pattern [index+1] pattern [index+2] = = L1L2L3) then
                {left_shift← next[i]; right_shift ←next[i]}
else if (pattern [0] pattern [1] = =L2L3) then
                {left_shift←m+1; right_shift←1}
else if (pattern [0] ==L3) {left_shift←length (pattern) +2; right_shift←2}
end  if
end of algorithm
```

Fig. 1. Algorithm for computing shift-left and shift-right failure functions

### *3.2. Search for Sequence Pattern in the Database*

This phase consists of searching for the sequence pattern over a specific attribute or column in given database. Initially the pattern is aligned with left end and right end of the input database. Algorithm scans concurrently from both sides of database to find the presence of user defined sequence pattern. Initially, algorithm starts by searching for presence of the sequence pattern from left end of the database.

While searching from left if a mismatch occurs, the sequence pattern window is shifted rightward by a shift value defined by the left shift failure function. The algorithm now performs the search process by scanning from the rightmost end of the database moving towards left and if a mismatch occurs; shift-right failure function is used to shift the pattern towards left. Algorithm terminates when the pattern is found or the left and right indices over cross each other.

**Step1:** At start the sequential pattern is aligned with left and right ends of the input database. To reduce the number of comparisons made, only the terminal elements of sequence pattern are compared. If the dead end elements match with pattern elements then we compare other pattern elements.

In an event of any mismatch, the algorithm moves to step2 otherwise the searching continues till the sequential pattern of interest is found. On success it reports the index position of the sequence pattern.

**Step2:** When a mismatch occurs from left end of the database while searching from the left, algorithm uses shift-left function to shift the sequence pattern making use of three attribute values of the input database which immediately appear after the aligned left sliding pattern. Similarly, shift value is computed using right failure function if a mismatch occurs from right end of database. This process continues until first occurrence of sequence pattern is found in the input database from either side or until both indices of sliding patterns over cross each other. The detailed algorithm for the same is given below in fig 2.

```
Left←0;                          //  index used from left
Right←n-m;                       //   index used from right
found←false;                     //    initial value
while (Left < = Right) do
       leftindex  ←length(pattern)-1;
       rightindex← length(pattern)-1;
if (Pattern [0] ==inputdB [Left])
  while (leftindex > 0) do
          if (Pattern[leftindex] ==inputdB [Left])
             leftindex←leftindex-1;
          else
             break;
   end while
if (Pattern [0] ==inputdB[Right])
    while(rightindex>0) do
           if (P[rightindex] ==T[rightindex+Right])
              rightindex←rightindex-1;
           else
              break;
     end while
if (leftindex = = 0) {sequence retrieved left end at: Left} ; exit from outer loop;
if(rightindex = = 0) {sequence retrieved at right end at: Right};  exit from outer loop;
Left← Left + shiftL (text substring (Left+m, Left+m+3));
Right←Right-shiftR (text substring (Right-3, Right));
if (Left > Right) {search is failure} exit from outer loop;
```

Fig. 2. Searching Algorithm

**4. Working Example**

| P1 | P2 | P3 | P4 |
|---|---|---|---|

where $P_1$ = tuple.value < tuple.previous.value
$P_2$ = tuple.value < tuple.previous.value ^ 40 < tuple.value < 50
$P_3$ = tuple.value > tuple.previous.value ^ tuple.value < 52
$P_4$ = tuple.value > tuple.previous.value.

and the operator '^' indicates logical AND operation.

Think of a scenario as in [8] where we are interested to find trends in the temperature for four consecutive days, i.e every possible instance where a sequence pattern consists of two immediate falls followed by two immediate hikes and the drops are such that the temperature to lie between 40°c and 50°c.

Also the first increase does not let the temperature move beyond 52°c.Here, we need to retrieve the sequence of Quintuple (A, B, C, D, E) with B.value < A.value, C.value < B.value, 40 <C.value<50, C.value<D.value, D.value<52, D.value<E.value holding well. The optimal set of constraints called as pattern elements are thus derived as shown above. The working of the algorithm is as explained below

**First iteration:** The search process begins by comparing pattern element at index=0 to the element of the inputdB at index=0. As there is a mismatch and also as all other cases violate, So left_shift failure function computes shift value as length(pattern)+ 3 = 7. The algorithm shifts sequence pattern to the right by 7 units. At the end of first iteration the algorithm makes the pattern indices from 0 to 3 aligned to indices of inputdB from 8 to 12.

**Second iteration:** Now the sequence pattern aligned at right end of the input database is considered for search process, pattern element at index 0 is compared with the database element at index 10. Since there is a mismatch, the immediate three attribute values to the left of the input database at indices 7, 8, 9 are considered to compute shift value using right_shift failure function. We stop the search process as there is a mismatch and start the search process from left end.

**Third iteration:** The elements of the sequence pattern from indices 0 to 3 are compared with the input database attribute values from indices 8 to 11 and it is a success. So stop and output the sequential pattern.

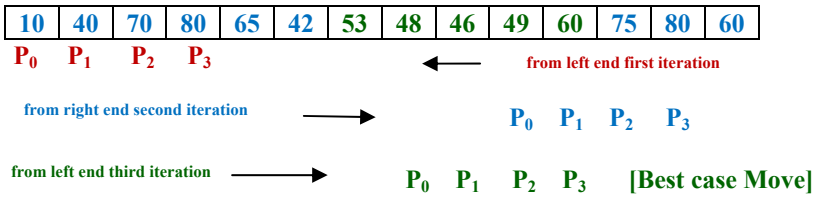


Fig. 3. Working Example

## 5. Conclusion

A sequential pattern mining algorithm with the pattern elements consisting of the user defined constraints is presented in this work. Failure functions are designed to handle the case of mismatch. The process of concurrent searching from the both ends of the database makes algorithm more efficient for the worst case situation when required sequence pattern is even at the end of the input. The reduction in the number comparisons done is visible from the best case move where algorithm shifts the pattern by a shift value greater than length of pattern when compared to algorithm used in [6].

**References**

[1] Knuth, D.E., J.H. Morris and V.R. Pratt. Fast pattern matching in strings. SIAM Journal of Computing. 6(2).1977, 323-350.
[2] Boyer, R. S. AND Moore, J. S. A fast string searching algorithm. Communications of the ACM. 20(10).1977, 762–772.
[3] R. Nigel Horspool. Practical fast searching in strings. Software Practice and Experience. John Wiley and Sons. (10) 1980, 501-506.
[4] T.BERRY AND S.RAVINDRAN A Fast String Matching Algorithm and Experimental Results, Proceedings of Prague Stringology
Club, workshop'99
[5] Handbook of Exact String Matching Algorithms.
[6] Expressing and Optimizing Sequence Queries in Database Systems.
http://www.cs.ucla.edu/~zaniolo/papers/todsjune04.pdf
[7] Wang, Y. and H. Kobayashi. High performance pattern matching algorithm for network security.6 (10). 2006, 83-87.
[8] Mjad Hudaib, Rola Al-Khalid, Dima Suleiman, Mariam Itriq and Aseel Al-Anani. A Fast Pattern Matching Algorithm Using Two
Sliding Windows, Journal of computer science. 4(5) 2008, 393-401.
[9] V.Radhakrishna, B.Phaneendra, V.Sangeeth Kumar. A Two Way Pattern Matching Algorithm Using Sliding Patterns. In the Proceedings of 3rd IEEE International Conference on Advanced Computer Theory and Engineering. (2) 2010, 666-670.
[10] V.Radhakrishna, C.Srinivas, Dr.C.V.Guru Rao. High Performance Pattern Search algorithm using three sliding windows. IJCET 3(2) 2012, 543-552. Journal Impact Factor: 3.9580 calculated by GISI.