Scalable Complex Pattern Search in Sequential Data

Leila Kaghazian Dennis McLeod Reza Sadri

Computer Science Department
University of Southern California
{kaghazia,mcleod}@usc.edu

Clenova Inc.
Irvine, California
reza@clenova.com

Abstract

Searching data streams has been traditionally very limited, either in the complexity of the search or in the size of the searched dataset. In this paper, we investigate the design and optimization of constructs that enable SQL to express complex patterns. In particular we propose the RSPS (recursive sequential pattern search) algorithm that inspired by the KMP (Knuth-Morris-Pratt) string matching algorithm and exploits the inter-dependencies between the elements of a sequential pattern to minimize repeated passes over the same data. Performance gains derived from a set of experiments and a sensitivity analysis for the RSPS are also discussed. Our experimental results demonstrate impressive speedup.

1. Introduction

Many applications in the commercial or scientific domains share the need for processing and analyzing sequential or stream data. Examples include analysis of data gathered from sensor networks, the stock market, telecommunications networks, seismic activity, and remote sensing. At times, the only feasible solution to understanding large volumes of data is to search for patterns of interest. This is an especially difficult task when the patterns of interest are complex in nature – in the sense that traditional constructs available in SQL may be unable to express these rich patterns. Facilities such as datablades have increased the expressive power of database query languages. However, many applications remain which need a more expressive language for describing their patterns of interest. Another challenge with most such complex applications is that data needs to be processed on the fly. The limited buffer needed for keeping the history of the time-series is thus another problem that needs to be addressed. An implementation of the pattern detection mechanism may be required which precludes keeping the entire history of the sequence in fast memory.

An extension of SQL with the ability to query time-series databases with more flexibility and power than Informix datablades [13] and SRQL [20] has been proposed before. Specifically in [23] Sadri *et al* introduce an extension of SQL, SQL-TS, to express sequential patterns, and study how to optimize search queries for this language. They exploit the inter-dependencies between the elements of a sequential pattern to minimize repeated passes over the same data.

While the technique outlined in [23] and [22] is powerful enough to find many types of patterns, it lacks the power necessary for expressing some key interesting queries. For instance, SQL-TS is not designed to search for patterns including nested stars (a recurring pattern inside another recurring pattern).

In this paper we extend the Optimal Pattern Search (OPS) algorithm [23] (termed RSPS) and present a general algorithm which gives SQL-TS the capability to look for more complex patterns such as nested-star patterns. RSPS provides a general framework to search for any pattern in SQL level. Preliminary results are encouraging and demonstrate the more expressive power of RSPS, and a dramatic speedup in operations.

In the next two sections, we briefly review SQL-TS and the OPS algorithm. We will not address them in detail as they have already been discussed in [23]. Our contribution here is to propose and explain the RSPS algorithm. Key research issues are opened up via our formulation of RSPS, as our overall goal is to address search for complex patterns, such an extension needs a radical modification.

2. SQL-TS

Structured Query Language for Time Series (SQL-TS), introduced in Sadri et al [23], adds a number of simple and useful constructs to SQL for specifying complex sequential patterns. SQL-TS is identical to SQL, except for the following additions to the FROM clause:

- a SEQUENCE BY clause specifying the sequencing attributes, and
- a CLUSTER BY clause specifying the grouping attributes, similar to GROUP BY. Each group indicates a separate sequence.

By way of an example, consider the following table of HTTP requests over the network:

```
CREATE TABLE network (
srcIP Varchar(15),
srcPort Varchar(5),
destIP Varchar(15),
destPort Varchar(5),
packet Integer (50),
date Date)
```

The following SQL-TS query (Example 1) finds the maximal periods in one day intervals, in which the number of incoming packets jumps more than 30%, and returns the source IP address and these periods:

Example 1: Using the FROM clause to define a pattern.

```
SELECT X.srcIP, X.date, Z.previous.date
FROM   network
       CLUSTER BY srcIP;
       SEQUENCE BY date
       AS (X,*Y, Z)
WHERE Y.packet > Y.previous.packet
        AND Z.previous.packet > X.packet*0.30
```

The AS clause – which in SQL is used mostly to assign aliases to table names – is used to specify a sequence of tuple variables from the specified table. Tuple variables from this sequence can be used in the WHERE clause to specify the conditions for expressing the pattern, and in the SELECT clause to specify the output.

A key feature of SQL-TS is its ability to express recurring patterns by using a star operator. However, the star operator can be applied only to simple patterns and not to complex patterns that contain sub-patterns. Our approach here will improve the power of SQL-TS by supporting recurring complex patterns, as detailed in Section 4.

3. Optimal Pattern Search (OPS)

The Optimal Pattern Search algorithm (OPS) was proposed by Sadri *et al* [23] for optimization of sequential queries in SQL-TS, via extending the KMP text matching algorithm [17]. The motivation behind development of OPS was the realization that finding sequential patterns in databases is somewhat similar to finding phrases in text. As such optimization techniques in OPS were inspired by string-matching algorithms. Possible choices that were considered as the basis for OPS were well-known string-matching algorithms with the best order of complexity in average cases. These included the Karp-Rabin algorithm [14], the Boyer-Moore pattern matcher [5] and the KMP algorithm [17]. Exhaustive experiments [28] demonstrated the performance superiority of KMP in most typical cases. Because of its good performance, and its independence from the alphabet size, and mostly because it generalizes to problems such as real-time string matching [12], KMP provides a natural basis for dealing with the more general problem of optimizing database queries on sequences. The generalization of KMP however presents difficult challenges: rather than searching for strings of letters (usually from a finite alphabet), we now have to search for sequences of structured tuples qualified by arbitrary expressions of propositional predicates involving arithmetic and aggregates.

Following is a brief summary of OPS: Given an input stream and a sequential query, suppose that while searching for the sequential pattern on an input stream, a mismatch occurs at the j^{th} position of the pattern. Speedup is achieved by tracking two items, *shift*(j) and *next*(j), that help reset the position trackers (i and j) to optimize values after the mismatch. *Shift*(j) determines how far the pattern should be advanced in the input, and *next*(j) determines from which element in the pattern the checking of conditions should be resumed after the shift. To compute *shift*(j) and *next*(j), OPS begins by capturing all the logical relations among pairs of the pattern elements using a positive precondition logic matrix θ, and a negative precondition logic matrix ϕ. These matrices are both of size m, where m is the length of the search pattern. The θ_{jk} and ϕ_{jk} elements of these matrices are only defined for $j \geq k$; thus they are lower-triangular matrices. θ_{jk} and ϕ_{jk} are defined as follows:

$$\theta_{jk} = \begin{cases} 1 & if \quad p_j \Rightarrow p_k \wedge p_j \neq F \\ 0 & if \quad p_j \Rightarrow \sim p_k \\ U & otherwise \end{cases}$$

$$\phi_{jk} = \begin{cases} 1 & if \quad \sim p_j \Rightarrow p_k \\ 0 & if \quad \sim p_j \Rightarrow \sim p_k \wedge p_j \neq T \\ U & otherwise \end{cases}$$

Here p_i is the predicate at location i. For instance, consider a pattern with the following predicates as its elements:

$p_1 = 5 < x < 9$

$p_2 = x > 10$

$p_3 = x < 20$

$p_4 = 21 < x < 50$

$p_5 = x < 2$

$p_6 = x < 5$

The matrices θ and ϕ for this pattern would be:

$$\theta = \begin{bmatrix} 1 & & & & & \\ 0 & 1 & & & & \\ U & U & 1 & & & \\ 0 & 1 & 0 & 1 & & \\ 0 & 0 & 1 & 0 & 1 & \\ 0 & 0 & 1 & 0 & U & 1 \end{bmatrix} \quad \phi = \begin{bmatrix} 0 & & & & & \\ U & 0 & & & & \\ 0 & 1 & 0 & & & \\ U & U & U & 0 & & \\ U & U & U & U & 0 & \\ U & U & U & U & 0 & 0 \end{bmatrix}$$

Logical relationships between whole patterns are derived from the matrices θ and ϕ, and *next* and *shift* are calculated accordingly.

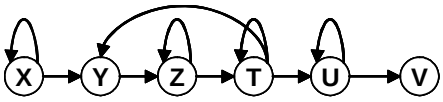


Figure 2: State model for Example 2

4. RSPS

An important advantage of the RSPS algorithm is that it can be easily generalized to handle input patterns which, in SQL-TS, are expressed using the star. In general, a star such as *Y denotes a maximal sequence of one or more tuples that satisfy all the applicable conditions. For example if predicate p_j is

$$t_i.price < t_{i-1}.price ,$$

then $*p_j$ matches sequences of records with decreasing prices. Now consider a more generalized example with the following predicates:

$$p_1(t) = t_i.price < t_{i-1}.price$$

$$p_2(t) = t_i.price > t_{i-1}.price .$$

In this case $*(*p_1,*p_2)$ matches the sequences of records with recurring patterns of decreasing prices followed by a period of increasing prices. Consider the following SQL-TS example.

Example 2: We are interested in finding the occurrence of the following pattern in Intel's stock price: an increasing period of time leading to repeated occurrence of a price between \$30 and \$40, followed by a period of decreasing price, followed by another period of increasing price, followed by another period of decrease leading to a price below \$25. The query written in SQL-TS is:

```
SELECT X.next.date, X.next.price,
       S.previous.date, S.previous.price
FROM quote
       CLUSTER BY name,
       SEQUENCE BY date
       AS (*X,*(Y,*Z,*T),*U,V)
WHERE
       X.name="Intel"
       AND X.price > X.previous.price
       AND 30 < Y.price
       AND Y.price < 40
       AND Z.price < Z.previous.price
       AND T.price > T.previous.price
       AND U.price < U.previous.price
       AND V.price < 25
```

Therefore our pattern predicates (on input tuple\ *t*) are:

```
p1(t) = (t.price > t.previous.price)
p2(t) = (30 < t.price <40)
p3(t) = (t.price < t.previous.price)
p4(t) = (t.price > t.previous.price)
p5(t) = (t.price < t.previous. price)
```

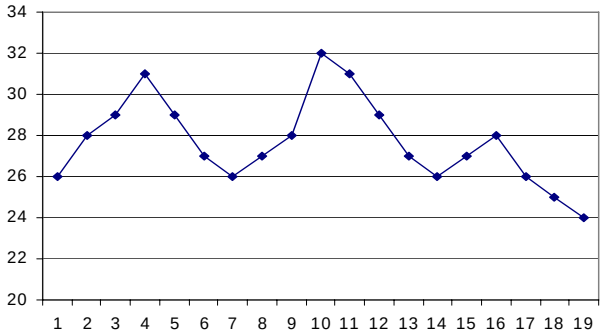


Figure 1: Illustration of nested recurring pattern in the input data in Example 2

```
p6(t) = (t.price < 25)
```

The calculation of logic matrices θ and ϕ remains unchanged in the presence of nested stars patterns; thus, the formulas given in Section 3 will still be used. However, the calculation of the arrays *next* and *shift* must be generalized for nested star patterns as described below.

At runtime we maintain an array of counters (one per pattern element) to keep track of the cumulative number of input objects that have matched the pattern sequence so far. For instance, if the first pattern element is a star that matched five elements in the input and the second pattern element is a non-star, matching only one input element, and the third element is a star matching two input elements we will have $count_1 = 5$, $count_2 = 6$, and $count_3 = 8$.

4.1 Run-time Support for Nested Stars

As mentioned earlier, at runtime we maintain an *array of counters* to keep track of the cumulative number of input objects that have matched the pattern sequence so far. Each element of this array is an array itself, since the star pattern can match different parts of the input stream in a single run. As such, we need a counter to keep track of the number of matched elements for each part.

For instance, suppose that the previous query is applied to our input stream with the following sequence for *t.price*:

26,28,29,31,29,27,26,27,28,32,
31,29,27,26,27,28, 26, 25, 24

Figure 1 illustrates the nested recurring pattern of the above dataset. After running the query, the array of counters will contain the following values:

$count_1 = 3$ $count_2 = 4{,}10$ $count_3 = 7{,}14$

$count_4 = 9{,}16$ $count_5 = 18$ $count_6 = 19$

4.2 Proposed Algorithm for Patterns with Nested Stars

As described above, some of the counters have more than one cumulative value. We shall employ these values in the algorithm proposed here for a pattern with nested stars.

RSPS Algorithm:

$OPS^*(i, j)$

$i \leftarrow 1 \quad j \leftarrow 1$

WHILE $((j \le m)\ and\ (i \le n))$

/* m is the length of the pattern and n is the length of the input data */

$R_j = \{k \mid \exists k\ s.t\ k \le j\ \&\ A(j,k) == 1\}$

/* R_j presents all possible nested star element of a sub-patterns start at k and end at j */

IF the current input element satisfies the pattern,

THEN

$i \leftarrow i+1$

IF R_j is empty

$j \leftarrow j+1$,

ELSEIF R_j is not empty and $\sim Sat(R_j, i)$ /* $Sat(R_j, i)$ returns the set of pattern elements */ $j \leftarrow j+1$ /* in R_j which satisfies i */

ELSE

$j \leftarrow \max(Sat(R_j, i))$

OTHERWISE (i.e. when the current input element doesn't satisfy the pattern)

IF R_j is empty or ($j == \max(R_j)$ and p_j is tested for the first time) then

- reset j (the index in the pattern) to *next(j)* and
- reset i (the index in the input) as follows:

$$i = i - j + \min\ (count\ (shift\ (j) + next\ (j) - 1))$$

IF R_j is not empty and $R_j - \max(R_j)$ is empty then

- $j \leftarrow j+1$

IF R_j is not empty and $j == \max(R_j)$ and $R_j - \max(R_j)$ is not empty then

- $j \leftarrow \max(R_j - \max(R_j))$

Figure 1 : RSPS Algorithm

We represent a pattern as a finite state model in which elements of the pattern are the states of the model. Stars and nested stars are coded in state transitions. Figure 2 illustrates a state diagram for Example 2. To develop the RSPS algorithm, the next step will be creating an adjacency matrix based on the state model of the pattern.

The following adjacency matrix presents the state model of Example 2:

$$A=\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

In the above matrix, the elements of patterns (X,Y,Z,T,U,V) were represented as (1,2,3,4,5,6) respectively. Hence $A(1,1) = 1$ means that p_1 (the first element of the pattern) is a star element. If $A(j,k)=1$ and $k < j$,then p_j is the last element of a nested star sub-pattern. For instance, in the above adjacency matrix, the 4th row represents the state T and $A(4,4) = 1$ means that T is a star element. Furthermore, since in the same row $A(4,2) = 1$, we conclude that T is the last element of a nested star sub-pattern, starting from Y. In the RSPS algorithm, for each row j, we define R_j which includes every k, ($k \le j$) such that $A(j,k) = 1$. For instance, in Example 2, $R_4 = \{2,4\}$. In Figure 1 we describe the RSPS algorithm in detail.

The difference between state models in OPS and RSPS is that the RSPS model may include right to left transitions; however in the OPS state model we only face left to right or self loop transitions. Note that in both OPS and RSPS, left to right transitions are only between adjacent states.

To complete the RSPS algorithm, we must now specify the computation of *shift*(j) and *next*(j) in the presence of nested stars.

4.2.1 Finding *shift* and *next* for the Nested Star Case

Consider the following sample graph based on the matrix θ (excluding the main diagonal):

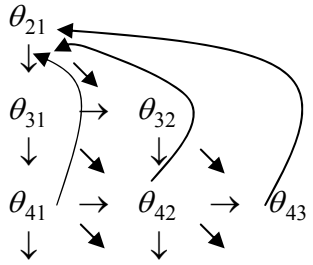


Figure 2: Matrix θ

The entry θ_{jk} in our matrix correlates pattern predicates p_j with p_k, $k < j$, when these are evaluated on the same input element. Therefore, we can picture the simultaneous processing of the input on the original pattern, and on the same pattern shifted back by $j - k$. Thus the arcs between nodes in our matrix above show the combined transitions in the original pattern and in the shifted pattern. In particular, consider θ_{jk} where neither p_k nor p_j are star predicates; then after success in p_j and p_k, we have a transition to p_{j+1} in the original pattern, and to p_{k+1} in the shifted pattern. This transition is represented by an arc $\theta_{jk} \rightarrow \theta_{j+1,k+1}$. However, if p_j is not a star predicate, while p_k is, then the success of both will move p_k, to $p_{k,+1}$, but leave p_j unchanged. This is represented by the arc $\theta_{jk} \rightarrow \theta_{j,k+1}$.

Here is an example to clear the concept. Suppose we have a pattern with 6 elements p_1 through p_6. We wish to evaluate the possible transition from the fifth element of the original pattern and the third element of the shifted pattern by two simultaneously on the same input element. In other words we are to evaluate the third and the fifth elements of the pattern on the same input element. Now assume that p_5 is a star element and p_3 is not a star element. Therefore, the possible transitions form θ_{53} would be to θ_{54} and θ_{64}. These transitions are illustrated in the above matrix as follow:

$$\theta_{53} \rightarrow \theta_{54}$$

$$\theta_{63} \searrow \theta_{64}$$

Figure 3: Example of transition in matrix θ

In the nested star situation, there is another possible transition which is a back edge when the last element of the nested star sub-pattern satisfies the previous input element but not the current one. In this case, before going forward to match the current input element with the next pattern element, the algorithm evaluates the input element against the first element of the nested star sub-pattern. The graph will thus have a back edge.

In general, we note that only some of the arcs listed in the matrix above represent valid transitions and should be considered. The set of valid transitions also depends on the values of θ. In particular, since all the predicates in the pattern must be satisfied by the shifted input, every $\theta_{jk} = 0$ entry must be removed with all its incoming and departing arcs: we only retain entries that are either 1 or U.

The directed graph produced by this construction will be called the *Implication Graph* for pattern sequence P, and will be denoted by G_p. For each value of j this graph must be further modified with entries from ϕ to account for the fact that j^{th} element of the pattern failed on the input. This is done via replacing the j^{th} row of G_p (i.e., the row that starts with $\theta_{j,1}$) with the j^{th} row of matrix ϕ, and removing all rows and arcs after j. In addition we recompute the arcs from row $j - 1$ to row j according to the new values of elements in row j.

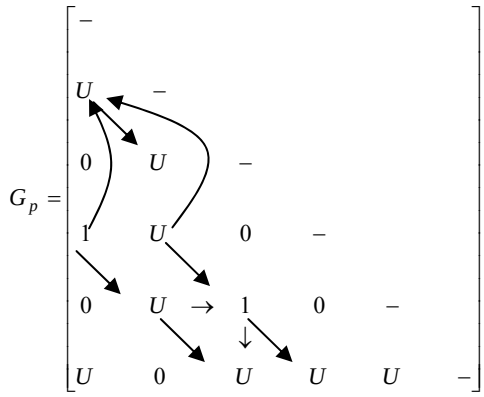
Thus, if element k is a star, there are up to two arcs from $\theta_{j-1,k}$ to row j: one to ϕ_{jk} and one to $\phi_{j,k+1}$. If element k is not a star, then there will be an arc only from $\theta_{j-1,k}$ to row j that goes to $\phi_{j,k+1}$. Furthermore, all the original G_p entries in rows up to and including $j - 1$ will remain unchanged, and so will all arcs leading to entries in these rows.

Again we assume that the end nodes of the arcs are either U or 1; but when such nodes are 0 the incoming arcs will be dropped. The resulting graph will be called the Implication Graph for pattern element j, denoted G_p^j; this graph will be used to compute *shift*(j) and *next*(j).

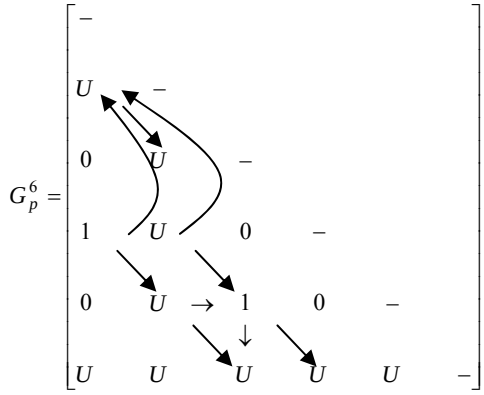
Let us consider Example 2 once more and calculate the matrices θ and ϕ:

$$\theta = \begin{bmatrix} 1 & & & & & \\ U & 1 & & & & \\ 0 & U & 1 & & & \\ 1 & U & 0 & 1 & & \\ 0 & U & 1 & 0 & 1 & \\ U & 0 & U & U & U & 1 \end{bmatrix} \qquad \phi = \begin{bmatrix} 0 & & & & & \\ U & 0 & & & & \\ U & U & 0 & & & \\ 0 & U & U & 0 & & \\ U & U & 0 & U & 0 & \\ U & U & U & U & U & 0 \end{bmatrix}$$

Since p_1, p_3, and p_5 are star predicates, p_4 is a nested star predicate, and p_2 and p_6 are not star predicates, the Implication Graph for our pattern sequence is as follows:



We now assume that we have a mismatch in presence of the current input data and the sixth element of the pattern. We thus need to construct G_p^6. We do so by replacing row 6 of G_p with the row 6 of ϕ and update the outgoing arcs from the row 5 to the new row 6. We will thus have:



Consider now the node θ_{41} in this graph. Observe that there are several paths consisting of either 1 node or *U* nodes that take us to nodes in the last row of the matrix. Therefore, the input shifted by 4 can succeed along any of these paths. However, there is no path to the last row starting from node θ_{31}: thus, 3 is not a possible shift. Also there are no paths to the last row starting from θ_{21} and θ_{11}; thus shifts of size 2 and 1 can never succeed. Therefore, we conclude that *shift*(6) = 3.

4.2.2 Computation of *shift(j)* and *next(j)* from G_p^j

A general definition of *shift*(*j*) is as follows:

Shift: *Let P denote the search pattern, and let* $\sigma(j) = \{s \mid \exists$ *a path from* $\theta_{s+1,1}$ *to a node in the last row of* $G_p^j\}$. Then :

- If the set $\sigma(j)$ is not empty, then $shift(j) = \min(\sigma(j))$.
- If the set $\sigma(j)$ is empty and $\phi_{j1} \neq 0$ then *shift*(*j*) = *j* –1.
- If the set $\sigma(j)$ is empty and $\phi_{j1} = 0$ then *shift*(*j*) = *j*.

Next: Multiple paths leading to the last row were acceptable for *shift*, but they are not acceptable for *next*, since this must return a value that uniquely determines the point from which the search must be resumed. Therefore, let us say that a node in our G_P^j graph is *deterministic* if there is exactly one arc leaving this node, and the end-node of this arc has value 1 (a deterministic node cannot therefore take us to a *U* node or to several 1 nodes). Thus, we start from $\theta_{shift(j)+1,1}$, and if this is not deterministic, then we set $next(j) = 1$. Otherwise, we move to the unique successor of this deterministic node and repeat the test. When the first non-deterministic node is found in this recursive process, *next*(*j*) is set to the value of its column. If the search takes us to the last row in G_P^j, that means that none of the input elements previously visited needs to be tested again: thus $next(j) = j - shift(j)$.

For the example at hand, there is a non-zero path from node θ_{41} to ϕ_{63}, thus *shift*(6) = 3. We now consider θ_{41} = 1 and see that this is not a deterministic node, since there is more than one arc leaving the node: one back edge to θ_{21} and one to θ_{52}.Thus, we conclude that $next(6) = 1$.

Despite the fact that the Implication Graph for RSPS may have some back edges, the computation for the *shift*(*j*) and *next*(*j*) is based on the same formula as the star algorithm. Suppose that there is a path from $\theta_{j-y,1}$ to the last row of the G_P^j. Also assume that there is a back edge from $\theta_{j-2,1}$ to the $\theta_{j-y,1}$ and there is a path from $\theta_{j-2,1}$ to the last row. Thus

$$\sigma(j) = \{j - y, j - 2\}(y > 2)$$

and

$$Shift(j) = min(\sigma(j))$$

Therefore the existence of a back edge in the Implication Graph does not have any impact on the calculation of *shift*(*j*) and therefore *next* (*j*).

Company Ticker	Sequence Length	RSPS Speedup (Example 3)	Number of Matches	OPS * on 100 simulated queries similar to Example 3				RSPS Speedup (Example 4)	Number of Matches
				Mean	Min	Max	Stdv		
DELL	4169	3.94	0	28.03	6.64	73.61	22.22	3.12	14
EBAY	1615	3.28	0	11.36	2.79	29.53	7.25	3.00	6
IBM	10863	20.08	2	66.43	11.16	139.90	44.60	12.32	54
GE	10863	16.21	4	58.55	9.27	150.94	43.59	9.35	78
COKE	3743	9.77	0	48.84	8.57	154.52	40.00	7.55	20
PEPSI	1480	8.35	2	28.55	8.59	62.80	15.75	7.68	12
SONY	5520	11.2	1	27.58	4.89	85.16	20.44	8.20	29
WMART	8204	8.04	3	64.94	9.35	214.27	55.72	5.04	72
DIJ	6000	92	1	86.93	37.11	201.20	43.97	85.22	14

Table 1: RSPS performance for selected companies for a given query (Example 3)
Gray area illustrates RSPS performance on 100 simulated queries similar to Example 3

5. Experimental Results

We perform a comparison here between RSPS, OPS and naïve search. We couldn't compare our results to any other approaches as there was no other pattern search algorithm in the same context. Even OPS was not able to handle the patterns with the nested stars.

To assess performance, we count the number of passes over the same input element while tested against a pattern element for both algorithms. The speedups obtained range from modest (simple search pattern without any recurring sub-pattern), to dramatic (more than two orders of magnitude obtained on complex patterns found in actual applications). We run RSPS over two set of datasets: stock market data and network data.

5.1 Stock Market Data

In stock market, there are a set of common chart patterns that can be very useful for technical analysis. Examples of such chart patterns are Double Bottom (two consecutive local minima that are roughly equal, with a moderate peak in between), Triple Top (three equal highs followed by a break below specific price) and Ascending Triangle (bullish formation that usually forms during an uptrend as a continuation pattern). In the following we show the power of RSPS in finding those patterns.

For instance, we ran the following patterns which reflex the search for a set of repeated consecutive relaxed double bottom in stock market data for a given company.

Example 3:

```
SELECT X.NEXT.date, X.NEXT.price,
        S.previous.date, S.previous.price
FROM company
        SEQUENCE BY date
        AS *(X,*Y, *Z, *T, *U, *V, *W, *R, S)
WHERE X.price >= 0.98 *X.previous.price
        AND Y.price < 0.98 *Y.previous.price
        AND 0.98*Z.previous.price < Z.price
        AND Z.price < 1.02*Z.previous.price
        AND T.price > 1.02 * T.previous.price
        AND 0.98*U.previous.price < U.price
        AND U.price < 1.02*U.previous.price
        AND V.price < 0.98 * V.previous.price
        AND 0.98*W.previous.price < W.price
        AND W.price < 1.02*W.previous.price
        AND R.price > 1.02*R.previous.price
        AND S.price <= 1.02*S.previous.price
```

We also ran RSPS for a similar query to Example 4 as following:

Example 4:

```
SELECT X.NEXT.date, X.NEXT.price,
        S.previous.date, S.previous.price
FROM company
        SEQUENCE BY date
        AS (X,*Y, *Z, *T, *U, *V, *W, *R, S)
```

Note that there is no (*) in front of the pattern in the last example. Table 1 compares the OPS and RSPS speedups for these patterns. As it is illustrated in Table 1 the speedups we obtained from running several queries were quiet dramatic (faster by up to a factor of 100).

The following are interesting observations from our experiment.

- RSPS speedup depends on the nature of the pattern query and the input itself. More inter-dependencies in between pattern elements make it possible to gain more speedup through RSPS.
- RSPS improves search speed even when there is no match for a given query. This case indeed is very interesting to study, when we want to make sure there is no occurrence of a given pattern query in a sequence or data stream.
- When there is no inter-dependency between the pattern elements, the RSPS speedup gets close to naïve search.

Data	Port #	Sequence Length	RSPS Speedup (Triple Bottom)	RSPS Speedup (Double Top)
Load	3	8000	13.0	24.6
Packet Rate	4	8000	21.6	24.1
Packet Collision	16	8000	4.4	13.5

Table 2: RSPS performance of network data

- RSPS pattern generalization provide a simple mechanism to relax the query when there is no occurrence of the pattern in the data.

To evaluate RSPS power, we employed a simulator with the capability of making complex queries around a given seed query. In our simulator, a user has the capability to modify the number of elements in the query, the length of the query and parameters in each element of the pattern. For instance assume user is interested in Example 4. We treat this query as seed query and make a set of queries around this query. We can modify Example 3 by changing numbers to parameters as following:

Example 3 (General):

```
SELECT X.NEXT.date, X.NEXT.price,
        S.previous.date, S.previous.price
FROM company
        SEQUENCE BY date
        AS *(X,*Y, *Z, *T, *U)
WHERE X.price >= A *X.previous.price
        AND Y.price < B *Y.previous.price
        AND C *Z.previous.price < Z.price
        AND Z.price < D *Z.previous.price
        AND T.price > E * T.previous.price
        AND U.price <= F *U.previous.price
        AND V.price < G * V.previous.price
        AND I*W.previous.price < W.price
        AND W.price < J*W.previous.price
        AND R.price > K*R.previous.price
        AND S.price <= L*S.previous.price
```

Now user can pick part of this query, drop or add (*) and change parameters (A,B ,…) . The following is a sample of modified version of Example 3.

Example 3 (modified):

```
SELECT X.NEXT.date, X.NEXT.price,
        S.previous.date, S.previous.price
FROM company
        SEQUENCE BY date
        AS *(X,*Y, *Z, *T)
WHERE X.price >= 0.90 *X.previous.price
        AND Y.price < 0.88 *Y.previous.price
        AND 0.98*Z.previous.price < Z.price
        AND Z.price < 1.05*Z.previous.price
        AND T.price > 1.05 * T.previous.price
        AND V.price < 0.97 * V.previous.price
        AND 0.96*W.previous.price < W.price
        AND W.price < 1.01*W.previous.price
        AND R.price > 1.03*R.previous.price
        AND S.price <= 1.02*S.previous.price
```

We ran the simulator to generate 100 queries and ran RSPS to find the pattern. The *mean, min, max* and *variance* of the RSPS speedup over naïve search are illustrated in Table 1. As it shows in Table 1 RSPS performance may varies dramatically (for instance form 200 to 37 in DIJ data) due to the query pattern.

5.2 Network Data

Understanding the nature of traffic in high-speed, high-bandwidth communications is essential for engineering and performance evaluation. Hence, finding patterns is an important essential for modeling the network behavior. Examples of these patterns are similar to stock market including Triple Bottom (three equal lows followed by a breakout above a certain level).

For this experiment we exploited a sample of network data. There are 16 ports on the routers that connect to 16 links, which in turn connect to 16 Ethernet subnets (Sn). Note that traffic has to flow through the router ports in order to reach the 16 subnets. There are three variables:

- *Load*: a measure of the percentage of bandwidth utilization of a port during a 10 minute period.
- *Packet Rate*: a measure of the rate at which packets are moving through a port per minute.
- *Collision Rate*: a measure of the number of packets during a 10 minute period that have been sent through a port over the link but have collided with other packets.

Data has collected for 18 weeks, from '94 to '95. There are 16,849 entries, representing measurements roughly every 10 minutes for 18 weeks. Figure 6 illustrates an example of collected *packet rate* data. For this experiment we illustrate the RSPS performance for on ports #3, #4, and #16 for *load, packet rate and packet collision* respectively.

Similar to stock market data we ran the simulator to generate 100 queries for a *triple top* query and ran RSPS to find the pattern. The *mean* and *variance* of the RSPS speedup over naïve search are illustrated in Table 1. As it is illustrated in Table 2 the speedups we obtained from running several queries were up to 25 times. Due to the space limit we only show the result of running RSPS on selected ports with better speedup. Similar to stock market data the RSPS performance may varies radically due to the pattern query and the data itself.

6. Related Work

Sequential pattern search is an important problem with broad applications, including the analysis of customer purchase behavior, web access patterns, scientific experiments, disease treatments, patient database, natural disasters, DNA sequences, network data analysis etc. Such problems have attracted researchers from different communities.

A major portion of research in this area has focused on discovering frequent patterns in sequential data (such as time series). The main focus of these works is to discover frequent patterns through approximation, transformation [1] and statistical inference. See for example, the approach taken by artificial intelligence researchers [15] [25] [9]. In the database context, where input data is usually much larger, the problem has been studied in a number of recent papers [2, 3, 9, 18, 26]. We are not looking for frequent patterns in time series; rather we are interested in exact match of a pattern in the SQL-TS level.

Recently there has been a great interest in processing streaming data. STREAM [4, 27] is a data stream processing project whose focus is on computing approximate results and to understand how to efficiently run queries in a bounded amount of memory. The Aurora [6] system allows users to specify quality-of-service requirements for queries, and then uses those specifications to determine how and when to shed load. Other recent research has focused on developing algorithms to perform specific functions on sequenced data. Gehrke et al. [10] considers the problem of computing correlated aggregate queries over streams, and presents techniques for obtaining approximate answers in a single pass.

Yang et al. [29, 30] discusses data structures for computing and maintaining aggregates over streams. Sadri et al. [23] propose SQL-TS, an extension of the SQL language to express sequence queries over time-series data. Finally, there has been a spate of work on this topic more recently, especially from the group at IIT-Bombay [8, 11, 21, 24]. Multi-query optimization typically shares relational sub expressions that appear in the plans of multiple (snapshot) queries. The Telegraph and TelegraphCQ [7] project have developed a suite of novel technologies for continuously adaptive query processing and on meeting the challenges that arise in handling large streams of continuous queries over high-volume, highly-variable data streams. Our proposed RSPS algorithm is a pattern detection mechanism that isn't bound to keeping the whole history of the sequence and trying to optimize the search by exploiting the inter-dependencies between the elements of a sequential pattern to minimize repeated passes over the same data.

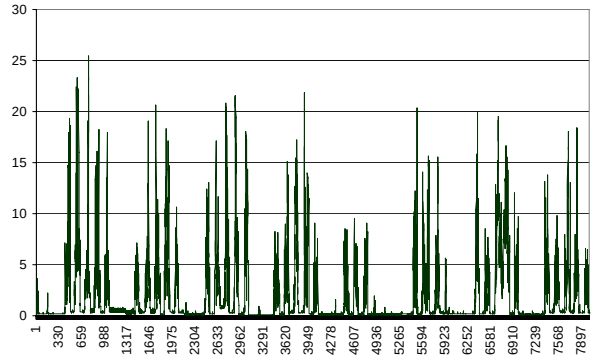


Figure 6: Example of packet rate for a given port

7. Future work and conclusion

In this paper, we described the RSPS algorithm which gives us a very expressive powerful tool to look for complex nested recurring patterns in sequential databases. By exploiting the inter-dependencies between the elements of a sequential pattern, we minimize repeated passes over the same data which impressively speedups the search process.

As RSPS provides a general framework to search for any pattern in SQL level, many applications in different domains could be beneficiaries of this powerful technique.

We are currently investigating to get even more speedup by employing statistical learning techniques to RSPS.

Our final goal is twofold: first, extending RSPS algorithm to be able to search in multidimensional data. Success in this matter will give us the ability to look for complex patterns in multidimensional domains such as images, trees and graphs. Second, considering that most streaming applications have a limited buffer size for the transient data, in order to make RSPS applicable to streaming applications, we need to address the buffer size issue. Since RSPS minimizes repeated passes over the same data by precomputing *shift(j)* and *next(j)*, it is obvious that in every pass we can remove some of the input data from the buffer and read the same amount of new incoming input data to the buffer. Our goal at this stage is to find an optimal buffer size along with the cost of processing. We are also looking to combining the approximation and sampling techniques to RSPS algorithm to look for complex patterns in stream data with even more speedup.

8. Acknowledgments

This research was supported in part by the Integrated Media Systems Center, A National Science Foundation Engineering Research Center, Cooperative Agreement No. EEC-9529152, and in part by NASA's Computational Technologies Project - portions of this work were carried out by the Jet Propulsion Laboratory, California Institute of Technology under contract with NASA, and in part by

the United States Department of Homeland Security through the Center for Risk and Economic Analysis of Terrorism Events (CREATE), grant number EMW-2004-GR-0112; however, any opinions, findings, and conclusions or recommendations in this document are those of the author(s) and do not necessarily reflect views of the U.S. Department of Homeland Security.

9. References

1. Agrawal, R., Faloutsos C. and Swami, A., "Efficient similarity search in sequence databases", *Proc. fourth International Conference on Foundataions of Data Organization and Algorithm*. 1993.
2. Agrawal, R., Lin, K. I., Sawheny, H. R., and Shim, K. *Fast similarity search in the presence of noise, scaling and translation in time series databases*. in *VLDB*. 1995.
3. Agrawal, R., Srikant, R. *Mining sequential pattern*. in *ICDE*. 1995. Taiwan.
4. Babcock, B., Babu, S., Datar, M., Motawani, R., and Widom, J. *Models and issues in data stream systems*. in *In Proceedings of the Twenties ACM SIGACT-SIGMOD-SIGART,* 2002.
5. Boyer, R.S.a.M., J. S., *A fast string searching algorithm.* Communications of the Association for Computing Machinery, 1977. **20**(10): p. 762-772.
6. Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N. and Zdonik, S. *Monitoring Streams: A New Class of Data Management Applications.* in *In proceedings of (VLDB'02),*. 2002. Hong Kong, China.
7. Chandrasekaran, S., Cooper,O., Deshpande, A., Franklin, M., Hellerstein, J., Hong, W., Krishnamurthy, S., Madden, S., Raman V., Reiss, F., Shah, M. *TelegraphCQ: Continuous Dataflow Processing for an Uncertain World*. in *CIDR*. 2003.
8. Dalvi, N., Sanghai, S., Roy, P., Sudarshan, S., Pipelining. *Multi-Query Optimization*. *PODS*. 2001.
9. Das, G., Lin, K., Mannila, H., Renganathan, G. and Smyth, P. *Rule discovery for time series*. in *KDD*. 1995. New York City, New York.
10. Gehrke, J.E., Korn, F., and Srivastava, D. *On Computing Correlated Aggregates Over Continual Data Streams.* SIGMOD, 2001.
11. Gupta, A., Sudarshan, S., Viswanathan, S. *Query Scheduling in Multi Query Optimization*. in *IDEAS*. 2001.
12. Gusfield, D., *Algorithms on Strings, Trees and Sequences*. 1997: Cambridge University Press.
13. Informix Software, I., *Managing time-series data infinancial applications*. 1998.
14. Karp, R.a.R., M. O, *Efficient randomized pattern matching algorithms.* IBM Journal of Research and Development, 1987. **31**(2): p. 249-260.
15. Keogh, E., Smyth, P. *A Probabilistic Approach to fast pattern matching in time series databases*. in *KDD*. 1997.
16. Kira, K.a.R., L. *The feature selection problem: Traditional methods and a new algorithms.* in *AAAI*. 1992: MIT Press.
17. Knuth, D.E., Morris, J. H., and Pratt, V. R., *Fast pattern matching in strings.* SIAM Journal of Computing, 1977. **6**(2): p. 323-350.
18. Mannila, H., Toivonen, H., and Verkamo, A. I. *Discovering generalized episodes using minimal occurance*. in *KDD*. 1996. Portland, Oregon.
19. Perng, C., Wang, H., Zhang, S. R. and Parker, D.,. *Landmarks: A new model for similarity-based pattern querying in time series databases,*. ICDE 2000.
20. Ramakrishnan, R.e.a. *SRQL: sorted relational query language*. 1998. SSDBM.
21. Roy, P., Seshadri, A., Sudarshan, A., Bhobhe, S. *Efficient and Extensible Algorithms For Multi Query Optimization.* in *SIGMOD*. 2000.
22. Sadri, R., Zaniolo, C., Zarkesh, A. and Adibi, J., *Expressing and Optimizing Sequence Queries in* (TODS), 2004.
23. Sadri, R., Zaniolo, C., Zarkesh, A., Adibi, J. *Optimization of pattern matching Queries on Database Sequences*. in *PODS, Twentieth ACM SIGACT-SIGMOD*. 2001. Santa Barbara,USA.
24. Sellis, T., *Multiple Query Optimization.".* ACM TODS, 1988. **13**(1): p. 23-52,.
25. Smyth, P., *Clustering sequences using Hidden Markov Models.* Advances in Neural Information Processing, 1997.
26. Tsong-Li Wang, J., Chim, G., Marr, T.G., Shapiro, B. A., Shasha, D., and Zhang, K. *Combinatorial pattern discovery for scientific data: Some preliminary results*. in *SIGMOD*. 1994.
27. Widom, S.B.a.J., *Continuous Queries over Data Streams.* SIGMOD Record, 2001.
28. Wright, C.A., Cumberland, L., and Feng, Y., *A performance comparison between five string pattern matching algorithms*. 1998, ocean.st.usm.edu/cawright/patternmatching.html.
29. Yang, J., and Widom, J. *Temporal View Self-Maintenance*. in *ACM TODS*. 2000.
30. Yang, J., AND Widom, J. *Incremental Computation and Maintenance of Temporal Aggregates*. in *ICDE*. 2001.