Optimization of Sequence Queries in Database Systems

Reza Sadri Carlo Zaniolo
Computer Science Department
University of California
Los Angeles, CA 90095
reza|zaniolo@cs.ucla.edu

Amir Zarkesh Jafar Adibi
ZAIAS Technologies Corporation
5086 Avenida Oriente
Tarzana, CA 91356
azarkesh|jadibi@U4cast.com.

ABSTRACT

The need to search for complex and recurring patterns in database sequences is shared by many applications. In this paper, we discuss how to express and support efficiently sophisticated sequential pattern queries in databases. Thus, we first introduce SQL-TS, an extension of SQL, to express these patterns, and then we study how to optimize search queries for this language. We take the optimal text search algorithm of Knuth, Morris and Pratt, and generalize it to handle complex queries on sequences. Our algorithm exploits the inter-dependencies between the elements of a sequential pattern to minimize repeated passes over the same data. Experimental results on typical sequence queries, such as double bottom queries, confirm that substantial speedups are achieved by our new optimization techniques.

1. INTRODUCTION

Many applications require processing and analyzing sequential data. Examples include the analysis of stock market prices [3], meteorological events [9], and the identification of patterns of purchases by customers over time [1, 2]. These applications focus on finding patterns and trends in sequential data. The patterns of interest in actual applications range from very simple ones, such as finding three consecutive sunny days, to the more complex patterns used in datamining applications [1, 4, 6]. These applications have motivated researchers to extend database query languages with the ability of searching for and manipulating sequential patterns.

The time-series datablades [6] introduced by Informix provide a library of functions that can be called from an SQL query, and most commercial DBMSs support similar extensions. But datablades lack in expressive power, flexibility and integration with DB query languages; thus, DB researchers have been seeking time-series tools that are more powerful, more flexible, and more integrated with DB query languages. In particular, the PREDATOR system proposed an SQL extension called SEQUIN [15, 16, 14] for querying sequences. Then, SRQL [12] extended the relational algebra with sequence operators for sorted relations, and added constructs for querying sequences to SQL.

In this paper, we view sorted relations as sequences as in SRQL, but propose a new and more powerful SQL-like language for pattern searching, and *advanced techniques for optimizing queries* in such a language.

2. THE SQL-TS LANGUAGE

Our Simple Query Language for Time Series (SQL-TS) adds to SQL simple constructs for specifying complex sequential patterns. For instance, say that we have the following table of closing prices for stocks:

```
CREATE TABLE quote ( name Varchar(8),
                     date Date,
                     price Integer )
```

Now, to find stocks that went up by 15% or more one day, and then down by 20% or more the next day, we can write the SQL-TS query of Example 1:

EXAMPLE 1. *Using the* `FROM` *clause to define patterns*

```
SELECT X.name
FROM quote
    CLUSTER BY name
    SEQUENCE BY date
    AS (X, Y, Z)
WHERE Y.price > 1.15 * X.price
    AND Z.price < 0.80 * Y.price
```

Thus, SQL-TS is basically identical to SQL, but for the following additions to the `FROM` clause:

- A `CLUSTER BY` clause specifying that data for each stock is processed separately (i.e., as it were a separate stream.)
- A `SEQUENCE BY` clause specifying that the data must be traversed by ascending `date`. Figure 1 shows how the `SEQUENCE BY` and `CLUSTER BY` statements affect the input. Rows are grouped by their `CLUSTER BY` attribute(s) (not necessarily ordered), and data in each group are sorted by their `SEQUENCE BY` attributes(s). This is similar to SRQL, where we have `GROUP BY` and `SEQUENCE BY` attributes [12].

NAME	PRICE	DATE
...	...	...
INTC	$60	1/25/99
INTC	$63.5	1/26/99
INTC	$62	1/27/99
...	...	...
IBM	$81	1/25/99
IBM	$80.50	1/26/99
IBM	$84	1/27/99
...	...	...

Figure 1: Effects of `SEQUENCE BY` and `CLUSTER BY` on data

- The `AS` clause, which in SQL is mostly used to assign aliases to the table names, is here used to specify a sequence of tuple variables from the specified table. By `(X, Y, Z)` we mean three tuples that immediately follow each other. Tuple variables from this sequence can be used in the `WHERE` clause to specify the conditions and in the `SELECT` clause to specify the output.

Expressing the same query using SQL would require three joins and would be more complex, less intuitive, and much harder to optimize.

A key feature of SQL-TS is its ability to express recurring patterns by using a star operator. Take the following example:

EXAMPLE 2. *Find the maximal periods in which the price of a stock fell more than 50%, and return the stock name and these periods*

```
SELECT X.name, X.date AS start_date,
      Z.previous.date AS end_date
FROM quote
   CLUSTER BY name
   SEQUENCE BY date
   AS (X, *Y, Z)
WHERE Y.price < Y.previous.price
   AND Z.previous.price < 0.5 * X.price
```

In SQL-TS, each tuple is viewed as containing two additional fields that refer to the `previous` and the `next` tuple in the sequence within the same cluster. Thus, for instance `Z.previous` (`X.next`) delivers the last tuple (the first tuple) in the $*$`Y` sequence, and `Z.previous.date` is the `date` of this last tuple (the SQL3 syntax `Z.previous` $\rightarrow$ `date` is also supported). Here the star construct $*$`Y` is used to specify a sequence of *one or more* `Y`'s of decreasing price, as per the condition the condition `Y.price < Y.previous.price`. In general, a star denotes a sequence of one or more (not zero or more!) tuples that satisfy all applicable conditions in the where clause. Thus, `Z` here is the first tuple where the price of the stock is no longer smaller than the previous one. Constructs similar to the star have been proposed previously in several query languages, and their semantics is easily formalized using recursive Datalog programs [11]. Also observe that a left maximality condition in implicit in the SQL-TS semantics, meaning that when two overlapping sequences satisfy the query, we return only the one that starts first.

3. SEARCH OPTIMIZATION

Since SQL-TS is a superset of SQL, all the well-known techniques for query optimization remain available, but in addition to those, we find new optimization opportunities using techniques akin to those used for text searching. For instance, take the following example:

EXAMPLE 3. *Find companies whose closing stock price in three consecutive days was 10, 11, and 15.*

```
SELECT X.name
FROM quote
   CLUSTER BY name
   SEQUENCE BY date
   AS (X, Y, Z)
WHERE X.price =10 AND Y.price=11
      AND Z.price=15
```

The text searching algorithms by Knuth, Morris and Pratt (KMP), discussed below, provides a solution of proven optimality for this query [8, 18]. Unfortunately, the KMP algorithm is only applicable when the qualifications in the query are equalities with constants as those of Example 3

Therefore, in this paper, we extend the KMP algorithm to handle the conditions that are found in general queries—in particular inequalities between terms involving variables such as those in the next example.

EXAMPLE 4. *For IBM stock prices, find all instances where the pattern of two successive drops followed by two successive increases, and the drops take the price to a value between 40 and 50, and the first increase doesn't move the price beyond 52.*

```
SELECT X.date AS start_date, X.price
   U.date AS end_date, U.price
FROM quote
   CLUSTER BY name
   SEQUENCE BY date
   AS (X, Y, Z, T, U)
WHERE X.name='IBM'
     AND Y.price < X.price
     AND Z.price < Y.price
     AND 40 < Z.price < 50
     AND Z.price < T.price
     AND T.price < 52
     AND T.price < U.price
```

3.1 Searching Simple Text Strings

The KMP algorithm takes a sequence pattern of length m, $P = p_1 \ldots p_m$, and a text sequence of length n, $T = t_1 \ldots t_n$, and finds all occurrences of P in T. Using an example from [8], let *abcabcacab* be our search pattern, and *babcbabcabcaabcabcabcacabc* be our text sequence. The algorithm starts from the left and compares successive characters until the first mismatch occurs. At each step, the i^{th} element in the text is compared with the j^{th} element in the pattern (i.e., t_i is compared with p_j). We keep increasing i and j until a mismatch occurs.

j, i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
t_i	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c
p_j	a	b	c	a	b	c	a	c	a	b							
				$\Uparrow$													

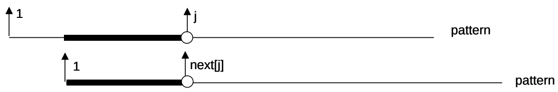

Figure 2: The meaning of $next(j)$

For the example at hand, the arrow denotes the point where the first mismatch occurs. At this point, a naive algorithm would reset j to 1 and i to 2, and restart the search by comparing p_1 to t_2, and then proceed with the next input character. But instead, the KMP algorithm avoids backtracking by using the knowledge acquired from the fact that the first three characters in the text have been successfully matched with those in the pattern. Indeed, since $p_1 \neq p_2$, $p_1 \neq p_3$, and $p_1p_2p_3 = t_1t_2t_3$ we can conclude that t_2 and t_3 can't be equal to p_1, and we can thus jump to t_4. Then, the KMP algorithm resumes by comparing p_1 with t_4; since the comparison fails, we increment i and compare t_5 with p_1:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
t_i	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c
j					1	2	3	4	5	6	7	8	9	10			
p_j					a	b	c	a	b	c	a	c	a	b			
												$\Uparrow$					

Now, we have the mismatch when $j = 8$ and $i = 12$. Here we know that $p_1 \ldots p_4 = p_4 \ldots p_7$ and $p_4 \ldots p_7 = t_8 \ldots t_{11}$, $p_1 \neq p_2$, and $p_1 \neq p_3$; thus, we conclude that we can move p_j four characters to the right, and resume by comparing p_5 to t_{12}. Therefore, by exploiting the relationship between elements of the pattern, we can continue our search without moving back in the text (i.e., without changing the value of i). As shown in [8], the KMP algorithm never requires backtracking on the text. Moreover, the index on the pattern can be reset to a new value $next(j)$, where $next(j)$ only depends on the current value, and is independent from the text. For a pattern of size m, $next(j)$ can be stored on an array of size m. (Thus this array can be computed once as part the query compilation, and then used repeatedly to search the database, and its time-varying content.)

The array $next(j)$ can be defined as follows:

1. Find all integers k, $0 < k < j$, for which $p_k \neq p_j$ and such that for every positive integer $s < k$, $p_s = p_{j-k+s}$ (i.e., $p_1 = p_{j-k+1} \wedge \ldots \wedge p_{k-1} = p_{j-1}$).
2. If no such k exists, then $next(j) = 0$ else $next(j)$ is the largest of these k's (yielding the least value of $j-k+1$).

This definition is clarified by Figure 2. The upper line shows the pattern, and the lower line shows the pattern shifted by k; the thick segments show where the two are identical. When no shift exists by which the shifted pattern can match the original one, we have $next(j) = 0$, and the pattern is shifted to the right till its first element is at position j, i.e., the current position in the text. In the KMP algorithm, this is the only situation in which the cursor on the input is advanced following a failure. (Of course, the input cursor is always advanced after success.)

The KMP Algorithm:

$j = 1;\ \ i = 1;$
while $j \leq m \ \wedge\ i \leq n$ do {
  while $j > 0 \ \wedge\ t_i \neq p_j$ do
    $j = next[j];$
  $i = i + 1;\ \ j = j + 1;$ }
if $i > n$ then failure
  else success;

The KMP algorithm is shown above. An efficient algorithm for computing the array $next$ is given in [8]. The complexity of the complete algorithm, including both the calculation of the $next$ for the pattern and the search of pattern over text, is $O(m + n)$, where m is the size of the pattern and n is the size of the text [8]. When success occurs, the input text $t_{i-m+1} \ldots t_i$ matches the pattern.

4. GENERAL PREDICATES

The original KMP algorithm can be used to optimize simple queries, such as that of Example 3, in which conditions in the `WHERE` clause are equality predicates and t is a tuple variable:

$$\begin{aligned} p_1(t) &= (t.price = 10) \\ p_2(t) &= (t.price = 11) \\ p_3(t) &= (t.price = 15) \end{aligned}$$

However, for the powerful sequence queries of SQL-TS we need to support:

1. **General Predicates:** In particular we need to support systems of equalities and inequalities such as those of Example 4 where we have the following predicates:

$$\begin{aligned} p_1(t) &= (t.price < t.previous.price) \\ p_2(t) &= (t.price < t.previous.price) \\ &\quad \wedge (40 < t.price < 50) \\ p_3(t) &= (t.price > t.previous.price) \\ &\quad \wedge (t.price < 52) \\ p_4(t) &= (t.price > t.previous.price) \end{aligned}$$

2. **Repeating pattern expressions:** The KMP algorithm assumes that the pattern consists of a fixed number of elements. To support queries such as that of Example 2, we need to optimize searches involving recurring patterns expressed by the star.
3. **More general objects:** In modern database systems we store many different types of objects, such as images, text, and XML objects, along with user-defined methods and predicates on these objects.

4.1 Optimized Pattern Search

In this section, we introduce the *Optimized Pattern Search (OPS) algorithm*, which is an extension the KMP algorithm. The OPS algorithm is directly applicable to the optimization of SQL-TS queries, since it handles the much more general conditions that occur in time series applications, including repeating patterns that can be expressed by the star construct.

Say that we are searching the input stream for a sequential pattern, and a mismatch occurs at the j-th position of the pattern. Then, we can use the following two sources of information to optimize our next steps in the search:

- All conditions for elements 1 through $j-1$ in the search pattern were satisfied by the corresponding items in the input sequence, and
- The condition for the j^{th} element in the search pattern was not satisfied by its corresponding input element.

Therefore, much as in the KMP algorithm, we can capture the logical relationship between the elements of the pattern, and then infer which shifts in the pattern can possibly succeed; also, for a given shift, we can decide which conditions need not be checked (since their validity can be inferred from the two kinds of information described above).

Therefore, we assume that the pattern has been satisfied for all positions before j and failed at position j, and we want to compute the following two items,

- $shift(j)$: this determines how far the pattern should be advanced in the input, and
- $next(j)$: this determines from which element in the pattern the checking of conditions should be resumed after the shift.

Observe that the KMP algorithm only used the $next(j)$ information. Indeed, for KMP, the search pattern was never shifted in the text (except for the case where $next(j) = 0$ and the pattern was shifted by j). The richer set of possibilities that can occur in OPS demand the use of explicit $shift(j)$ information. Furthermore, the computation for $next$ and $shift$ is now significantly more complex and requires the derivation of several three-valued logic matrices.

4.2 Implications Between Elements

The OPS algorithm 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 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 we have lower-triangular matrices of size m. We define:

$$\theta_{jk} = \begin{cases} 1 & \text{if} \quad p_j \Rightarrow p_k \quad \wedge \quad p_j \not\equiv F \\ 0 & \text{if} \quad p_j \Rightarrow \neg p_k \\ U & \text{otherwise} \end{cases}$$

$$\phi_{jk} = \begin{cases} 1 & \text{if} \quad \neg p_j \Rightarrow p_k \\ \emptyset & \text{if} \quad \neg p_j \Rightarrow \neg p_k \quad \wedge \quad p_j \not\equiv T \\ U & \text{otherwise} \end{cases}$$

We have added the terms $p_j \not\equiv F$ in definition of θ, and $p_j \not\equiv T$ in definition of ϕ, to make sure that the left side of the implication relationships are not equivalent to false, because in that case the value of the corresponding element in the matrix could be both 0 and 1. By excluding those

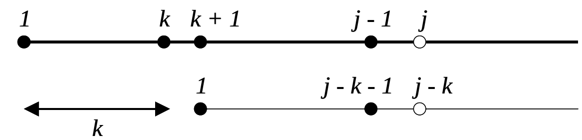

Figure 3: Shifting the pattern k positions to the right

cases, we have removed the ambiguity. Logic matrices θ and ϕ contain all the possible pairwise logical relations between pattern elements. For instance, for Example 4 we have:

EXAMPLE 5. *Computing the matrices θ and ϕ for Example 4*

$$\begin{array}{lll} p_2 \Rightarrow p_1 & \textit{therefore} & \theta_{21} = 1 \\ p_3 \Rightarrow \neg p_1 & \textit{therefore} & \theta_{31} = 0 \\ p_3 \Rightarrow \neg p_2 & \textit{therefore} & \theta_{32} = 0 \\ p_4 \Rightarrow \neg p_2 & \textit{therefore} & \theta_{42} = 0 \\ p_4 \Rightarrow \neg p_1 & \textit{therefore} & \theta_{41} = 0 \\ \neg p_4 \Rightarrow \neg p_3 & \textit{therefore} & \phi_{43} = 0 \end{array}$$

Therefore we have

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

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

¿From matrices ϕ and θ, we can now derive another triangular matrix S that describes the logical relationships between whole patterns. The S_{jk} entries in the matrix, which are only defined for $j > k$, are computed as follows:

$$S_{jk} = \theta_{k+1,1} \wedge \theta_{k+2,2} \wedge \cdots \wedge \theta_{j-1,j-k-1} \wedge \phi_{j,j-k}$$

Thus, say that the pattern was satisfied up to, and excluding, element j; then, $S_{jk} = 0$ means that the pattern cannot be satisfied if shifted k positions. Moreover, $S_{jk} = 1$ ($S_{jk} = U$) means that the pattern is certainly (possibly) satisfied after a shift of k. Figure 3 illustrates the situation. In calculating matrix S, we use standard 3-valued logic, where $\neg U = U$, $U \wedge 1 = U$, and $U \wedge 0 = 0$. For the example at hand we have:

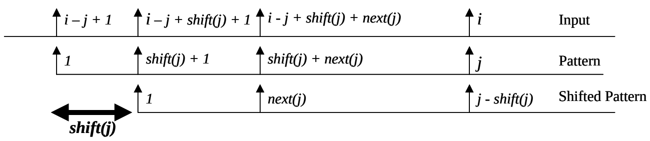

Figure 4: Next and Shift definitions for OPS

EXAMPLE 6. *Computing the matrix S for Example 5*

$$\begin{aligned}
S_{2,1} &= \phi_{2,1} = U \\
S_{3,1} &= \theta_{2,1} \wedge \phi_{3,2} = 1 \wedge U = U \\
S_{3,2} &= \phi_{3,1} = U \\
S_{4,1} &= \theta_{2,1} \wedge \theta_{3,2} \wedge \phi_{4,3} = 0 \\
S_{4,2} &= \theta_{3,1} \wedge \phi_{4,2} = 0 \\
S_{4,3} &= \phi_{4,1} = U
\end{aligned}$$

$$S = \begin{bmatrix} & & & \\ U & & & \\ U & U & & \\ 0 & 0 & U & \end{bmatrix}$$

We can now compute $shift(j)$, which is the least shift to the right for which the overlapping sub-patterns do not contradict each other (Figure 4). Thus, $shift(j)$ is the column number for the leftmost non-zero entry in row j of S. When all these entries are equal to zero, then a failure will occur for any shift up to j. In this case, we set $shift(j) = j$; thus, the pattern is shifted to the right till its first position coincides with the position immediately after the cursor in the text. More formally:

$$shift(j) = \begin{cases} j \quad \text{if} \quad \forall k < j,\ S_{jk} = 0 \\ \mathbf{min}(\{k \mid S_{jk} \neq 0\}) \quad \text{otherwise} \end{cases}$$

Thus, $shift(j)$ tells us how much the pattern can be advanced on the input before there is any chance of success. We can now compute $next(j)$ which denotes the element in the pattern from which checking against the input should be resumed (for elements before $next(j)$ the result is already known to be true). There are basically three case. The first case is when $shift(j) = j$, and thus the first element in the pattern must be checked next against the current element in the input. The second case is when $shift(j) < j$ and $S_{j,shift(j)} = 1$; In this case we only need to begin our checking from the element in the pattern that is aligned with the first input element after current input position—thus, $next(j) = j - shift(j) + 1$. The third case occurs, when neither of the previous cases hold; then the first pattern element should be applied to the input element $i - j + shift(j) + 1$; but if $\theta_{shift(j)+1,1} = 1$, then the comparison becomes unnecessary (and similar conditions might hold for the elements that follow). Thus, we set $next(j)$ to the leftmost element in the pattern that must be tested against the input. Figure 4 shows how this works. Now we can formally define *next* as follows:

1. if $shift(j) = j$ then $next(j) = 0$, else

2. if $S_{j,shift(j)} = 1$ then
   $next(j) = j - shift(j) + 1$, else

3. if neither of these conditions hold, then
   $next(j) = \mathbf{min}($
   $\{t \mid 1 \leq t < j - shift(j) \wedge \theta_{shift(j)+t,t} = U\}$
   $\cup\ \{j - shift(j) | \phi_{j,j-shift(j)} = U\})$

For the example at hand we have:

EXAMPLE 7. *Compute shift and next for Example 5*

$$\begin{array}{lll}
shift(1) = 1 & & \\
shift(2) = 1 & \text{since} & S_{21} \neq 0 \\
shift(3) = 1 & \text{since} & S_{31} \neq 0 \\
shift(4) = 3 & \text{since} & S_{41} = 0 \wedge S_{42} = 0 \\
 & & \wedge S_{43} \neq 0 \\
 & & \\
next(1) = 0 & \text{since} & shift(1) = 1 \\
next(2) = 1 & \text{since} & \phi_{21} \neq 1 \\
next(3) = 2 & \text{since} & \theta_{21} = 1 \wedge \phi_{32} \neq 1 \\
next(4) = 1 & \text{since} & \phi_{41} \neq 1
\end{array}$$

The calculation of arrays *shift* and *next* is done as part of query compilation. This is discussed in Section 6.

4.2.1 The Main Algorithm.

We can use the values stored in arrays *next* and *shift* to optimize the pattern search at run time. Consider a predicate pattern $p_1 p_2 \dots p_m$. Now, $p_j(t_i)$ is equal to one, when the i-th element in the input sequence satisfies a pattern element p_j; otherwise, it is zero.

The OPS Algorithm

$j = 1;\ \ i = 1;$
while $\ j \leq m\ \wedge\ i \leq n\ $ do {
  while $\ j > 0\ \wedge\ \neg p_j(t_i)\ $ do {
    $i = i - j + shift(j) + next(j);$
    $j = next(j);$ }
  $i = i + 1;\ \ j = j + 1;$ }
if $\ i > n$ then failure
  else success;

Here too, as in the KMP algorithm, *success* denotes that $t_{i-m+1} \dots t_i$ satisfies the the pattern. However, we see the following generalizations with respect to KMP:

- The equality predicate $t_i = p_j$ is replaced by $p_j(t_i)$ that tests if p_j holds for the i-th element in the input (i.e., the j-th tuple of the sorted cluster).
- When there is a mismatch, we modify both j and i, which, respectively, index the input and the pattern. The new value for j is $next(j)$ and the new value for i is $i - j + shift(j) + next(j)$.

For instance, we used the pattern in the query of Example 4 to search the following sequence:

55 50 45 57 54 50 47 49 45 42 55 57 59 60 57.

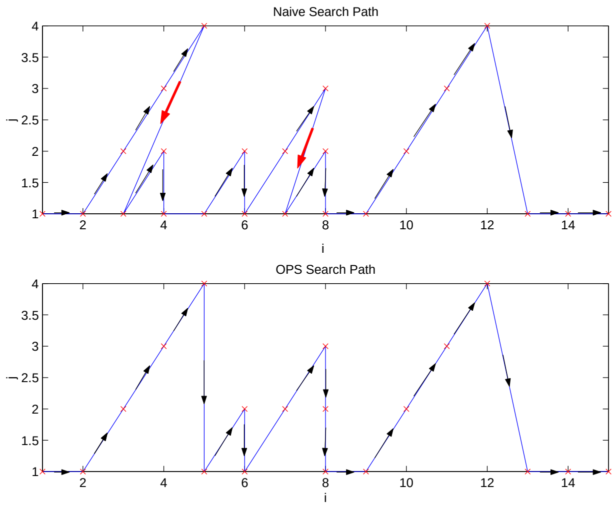

Figure 5: Comparison between path curve of the naive search (top chart) and OPS (bottom chart)

Figure 4.2.1 compares the evolution of the values of j and i for the naive algorithm and the OPS algorithm. Clearly, for the OPS algorithm, the backtracking episodes are less frequent and less deep, and therefore the length of the search path is significantly shorter.

5. DEALING WITH THE STAR

An important advantage of the OPS algorithm is that it can be easily generalized to handle recurrent input patterns which, in SQL-TS, are expressed using the star. For example, say that $*p$ is an element in our search pattern, where $p = t.price < t.previous.price$. Then $*p_j$ matches any sequence of records with decreasing prices.

The calculation of the logic matrices θ and ϕ remains unchanged in the presence of star patterns; thus, the formulas given in Section 4.2 will still be used. However, the calculation of the arrays $shift$ and $next$ must be generalized for star patterns as described next. Consider the following SQL-TS query:

EXAMPLE 8. *Find patterns consisting of a period of rising prices, followed by a period of falling prices, followed by another period of rising prices.*

```
SELECT X.name, FIRST(X).date AS sdate,
     LAST(Z).date AS edate
FROM quote
   CLUSTER BY name
   SEQUENCE BY date
   AS ( *X, *Y, *Z)
WHERE X.price > X.previous.price
   AND Y.price < Y.previous.price
   AND Z.price > Z.previous.price
```

Therefore, the three predicates that must be satisfied by the tuples, X, Y and Z, are as follows:

$$\begin{aligned} p_1(X) &= (X.price > X.previous.price) \\ p_2(Y) &= (Y.price < Y.previous.price) \\ p_3(Z) &= (Z.price > Z.previous.price) \end{aligned}$$

These will be called *star* predicates, because they are prefixed with a star in the 'from' clause of the query, which searches for the pattern: $*p_1(X),\ *p_2(Y), *p_3(Z)$.

To support efficient search on patterns with star, at runtime, we maintain an array of counters, one per pattern element. Each counter keeps track of the cumulative number of input tuples that have matched the pattern up to this element. For instance, say that we have the following sequence of values for $t.price$:

$$20 \quad 21 \quad 23 \quad 24 \quad 22 \quad 20 \quad 18 \quad 15 \quad 14 \quad 18 \quad 21$$

and let $count(j)$ denote the counter for the j-th element of the pattern. After matching the pattern with the text we have:

$\text{count}(1) = 4$	
$\text{count}(2) = 9$	since 5 elements satisfy p_2
$\text{count}(3) = 11$	since 2 elements satisfy p_3.

We update and use these counters at runtime while searching the input for sequences that satisfy the pattern. Therefore, for star patterns, our search algorithm is generalized as described next.

If the current input element *satisfies* the pattern then, we advance the input cursor to the next element, and if

1. *the current pattern element is not a star*, we advance the pattern cursor to the next element, otherwise
2. *the current element is a star* and we update count to count + 1 (and leave the cursor on the current pattern element).

If the current input element *does not satisfy* the pattern, then

1. if the current pattern element is a star predicate, which has already been satisfied by the previous input element, then we advance the pattern cursor and the input cursor to their next respective elements;
2. if the current pattern element is not a star predicate, or it is a star predicate which has not been tested on the previous input element, then we
   - reset j (the index in the pattern) to $next(j)$, and
   - reset i (the index in the input) to: $i - \text{count}(j-1) + \text{count}(shift(j) + next(j) - 1)$.

In the presence of stars, the compile-time computation for $shift(j)$ and $next(j)$ is more complex, and it is discussed next.

5.1 Finding $next$ and $shift$ for the Star Case

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

$$\begin{array}{ccccc}
\theta_{21} & & & & \\
\downarrow & \searrow & & & \\
\theta_{31} & \rightarrow & \theta_{32} & & \\
\downarrow & \searrow & \downarrow & \searrow & \\
\theta_{41} & \rightarrow & \theta_{42} & \rightarrow & \theta_{43} \\
\downarrow & \searrow & \downarrow & \searrow &
\end{array}$$

The entry θ_{jk} in our matrix correlates pattern predicates p_j with p_k, $k < j$, when the two 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 with the cursor shifted back by $j-k$. Thus, the arcs between nodes in our matrix above represent the combined transitions in the original pattern and in the shifted pattern. In particular, consider θ_{jk} where neither p_j nor p_k are star predicates; then after success in p_j and p_k, we 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 a star predicate, while p_k is not, then the success in both will move p_k to p_{k+1}, but leave p_j unchanged: this is represented by the arc $\theta_{jk} \rightarrow \theta_{j,K+1}$. In general, it is clear that only a subset of the arcs listed in the previous matrix represent valid transitions, and should be considered, and this set is further limited by the values of θ. In particular, since all the predicates in the pattern must be satisfied by the shifted input, every arc to and from a node that has value 0 can be discarded: we only keep those arcs where both end-points are 1 or U.

If we consider all possible transitions, we conclude that only the following arcs are valid from any given node in the graph, assuming that all its neighbors are nonzero nodes:

1. If both elements j and k of the pattern sequence are star predicates and $\theta_{jk} = U$, then we have three outgoing arcs from θ_{jk}: one to $\theta_{j+1,k}$, one to $\theta_{j+1,k+1}$ and one to $\theta_{j,k+1}$. Pictorially,

$$\begin{array}{ccc}
U & \rightarrow & \theta_{j,k+1} \\
\downarrow & \searrow & \\
\theta_{j+1,k} & & \theta_{j+1,k+1}
\end{array}$$

2. If both element j and element k of the pattern are stars and $\theta_{jk} = 1$, we have two outgoing arcs from θ_{jk}: one to $\theta_{j+1,k+1}$ and the other to $\theta_{j,k+1}$. Pictorially,

$$\begin{array}{ccc}
1 & & \theta_{j,k+1} \\
\downarrow & \searrow & \\
\theta_{j+1,k} & & \theta_{j+1,k+1}
\end{array}$$

   There is no arc to $\theta_{j,k+1}$, because $\theta_{j,k} = 1$; thus all input tuples that satisfy p_j must also satisfy p_k.

3. If both elements j and k of the pattern are non-star predicates, then we have only one arc from θ_{jk} to $\theta_{j+1,k+1}$. Pictorially,

$$\begin{array}{ccc}
\theta_{jk} & & \theta_{j,k+1} \\
 & \searrow & \\
\theta_{j+1,k} & & \theta_{j+1,k+1}
\end{array}$$

4. If element j of the pattern is a star predicate, but element k is not, then we have two arcs from θ_{jk}: one to $\theta_{j+1,k+1}$ and the other to $\theta_{j,k+1}$:

$$\begin{array}{ccc}
\theta_{jk} & \rightarrow & \theta_{j,k+1} \\
 & \searrow & \\
\theta_{j+1,k} & & \theta_{j+1,k+1}
\end{array}$$

5. If element k of the pattern is a star predicate but element j is not, then we have two arcs from θ_{jk}: one to $\theta_{j+1,k+1}$ and the other to $\theta_{j+1,k}$. Thus we have:

$$\begin{array}{ccc}
\theta_{jk} & & \theta_{j,k+1} \\
\downarrow & \searrow & \\
\theta_{j+1,k} & & \theta_{j+1,k+1}
\end{array}$$

For all the arcs shown above, we have assumed that their end nodes are either U or 1; however, when such nodes are 0, their incoming arcs will instead be dropped. The directed graph produced by this construction will be called the *Implication Graph* for pattern sequence P, and will be denoted as G_P.

The graph G_P has m nodes where m is the number of elements in the pattern. Now, for each value of $j \leq m$, the matrix representing G_P must be modified with entries from ϕ to account for the fact that the jth element of the pattern failed on the input. Therefore, we replace the jth row of G_P (i.e., the row that starts with $\theta_{j,1}$) with the jth row of matrix ϕ and remove rows greater than j.

We also update the arcs between elements in row $j-1$ and row j according to the new values of elements in row j. Basically, the rules previously discussed for arcs between θ nodes still hold now that the arcs lead to ϕ nodes. However, since we only want arcs that have U or 1 as their end-node, we eliminate all arcs leading to a node $\phi_{j,k}$ in the last row when this node has value 0. The resulting graph will be called the *Implication Graph for pattern element* j, denoted G_P^j.

The following SQL-ST query illustrates the computation of or matrices θ and ϕ. We want to find occurrences of the following four-period pattern in IBM's stock prices:

Example 9. *Find occurrences of the following four-period patterns in IBM's prices: (i) a period of increasing prices in the 30-40 range, followed by (ii) a period of decreasing prices, followed by (iii) another period of price increases moving the price into the 35-40 range, (iv) followed by a period of price decreases taking the price below 30.*

```
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, S)
WHERE
   X.name='IBM'
   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 35 < U.price
   AND U.price < 40
   AND V.price < V.previous.price
   AND S.price < 30
```

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

$$
\begin{array}{rcl}
p_1(t) & = & (t.price > t.previous.price) \\
p_2(t) & = & (30 < t.price < 40) \\
p_3(t) & = & (t.price < t.previous.price) \\
p_4(t) & = & (t.price > t.previous.price) \\
p_5(t) & = & (35 < t.price < 40) \\
p_6(t) & = & (t.price < t.previous.price) \\
p_7(t) & = & (t.price < 30)
\end{array}
$$

Observe that p_1, p_3, p_4, and p_6 are star predicates, and the others are not. Our matrices ϕ and θ are:

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

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

Since p_1, p_3, p_4, and p_6 are star predicates, and p_2 and p_5 are not, we will connect the elements of θ (after excluding the main diagonal), and obtain the following matrix G_P:

$$
\left[\begin{array}{ccccccccccccc}
- & & & & & & & & & & & & \\
U & & - & & & & & & & & & & \\
 & \searrow & & & & & & & & & & & \\
0 & & U & & - & & & & & & & & \\
1 & & U & & 0 & & - & & & & & & \\
\downarrow & \searrow & & \searrow & & & & & & & & & \\
U & & 1 & & U & & U & & - & & & & \\
 & \searrow & & \searrow & \downarrow & & & \searrow & & & & & \\
0 & & U & \rightarrow & 1 & & 0 & & U & & - & & \\
 & & & \searrow & \downarrow & \searrow & & & & \searrow & & & \\
U & & 0 & & U & & U & & 0 & & U & & -
\end{array}\right]
$$

Say now that we want to build G_P^6. We replace row 6 of G_P with row 6 of ϕ and update the paths from the 5th row to the 6th row according to new value. Thus, the graph G_P^6 is as follows:

$$
\left[\begin{array}{ccccccccccc}
- & & & & & & & & & & \\
U & & - & & & & & & & & \\
 & \searrow & & & & & & & & & \\
0 & & U & & - & & & & & & \\
1 & & U & & 0 & & - & & & & \\
\downarrow & \searrow & & \searrow & & & & & & & \\
U & & 1 & & U & & U & & - & & \\
\downarrow & \searrow & & & & \searrow & \downarrow & \searrow & & & \\
U & & U & & 0 & & U & & U & & -
\end{array}\right]
$$

Consider now the node θ_{41} in this graph. Observe that there are several paths nodes that take us to nodes in the last row of the matrix, through a succession of nodes whose values are either 1 or U. Therefore, an input shifted by 3 can succeed along any of these paths (with a shift of 3, the search resumes by comparing element $3+1$ in the input against element 1 in the pattern).

However, there is no path to the last row starting from node θ_{31}: thus, 2 is not a possible shift. Also there is not path to the last row starting from θ_{21}; thus a shift of size 1 will never succeed. Therefore, we conclude that $shift(6) = 3$.

In general, we define $shift(j)$ as follows:

DEFINITION 1. *Let P denote the search pattern, and let*

$$
\sigma(j) = \{s \mid \exists \text{ a path from } \theta_{s+1,1} \text{ to a node in the last row of } G_P^j \}.
$$

then,

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

Now, we can define $next$. 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 (thus a deterministic node cannot take us to an 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 we set $next(j) = j - shift(j)$.

For the example at hand, there is a non-zero path from node θ_{41} to ϕ_{61}, thus $shift(6) = 3$. We now consider $\theta_{41} = 1$ and see that this is not a deterministic node, since there more than one arc leaving the node. Thus, we conclude that $next(6) = 1$.

For the computation of $shift(j)$, we must find from which nodes in the first column the last row of G_P^j can be reached. Transitive closure algorithms can be used to identify the nodes in the first column connected with nodes in the last row. But for a pattern of length m, we have $m(m-1)/2$ nodes in our graph; thus classical algorithms for transitive closures, such as the Warshall algorithm, can have complexity $O(m^6)$. A better approach consists in using the inverse graph, extended with a root node that has arcs leading to each node in the last row of our matrix. Then, we can traverse this graph from the root, and among the visited nodes in the first column, select the one with the smallest row number (or return the information that this set is empty). The complexity of graph traversal is linear in the number of arcs in the graph. Thus, the computation of a single $shift(j)$ takes $O(m^2)$, in the worst case. After determining $shift(j)$, we compute $next(j)$ by following a linear path on the graph till we find a U, or a fork or we reach the last row. But, due to the orientation of its arcs, our graph cannot contain any path of length greater than $2 \times m$. Thus, the computation of $next(j)$ is linear in m. Therefore, the computation of all pairs $shift(j)$ and $next(j)$, for $1 \leq j \leq m$, has complexity $O(m^3)$.

6. IMPLEMENTATION

Elements of ϕ and θ are calculated based on the semantics of the pattern elements, particular inequalities between pattern elements. Several satisfiability and implication results in databases [5] are relevant to calculate the nodes of the θ and ϕ matrices, for classes of patterns that involve inequality. In our implementation, we used the algorithm by Guo, Sun and Weiss (GSW) [5] for computing implication and satisfiability of conjunctions of inequalities. In the computation of our ϕ and θ matrices, the implication algorithm is used to determine which nodes have a value of 1, and the satisfiability algorithm is used to determine the nodes that have value of 0. The GSW algorithm deals with inequalities of the form $X\ op\ C$, $X\ op\ Y$, and $X\ op\ Y + C$ where X and Y are variables, C is constant, and $op \in \{=, \neq, \leq, \geq, <, >\}$. Complexity of their algorithm is $O(|S| \times n^2 + |T|)$ for testing implication and $O(|S| + n^3)$ for testing satisfiability; n is the number of variables in S; $|S|$ and $|T|$ are the number of inequalities respectively contained in S and T. Given the limited number of variables and inequalities used in actual queries these compilation costs are quite reasonable.

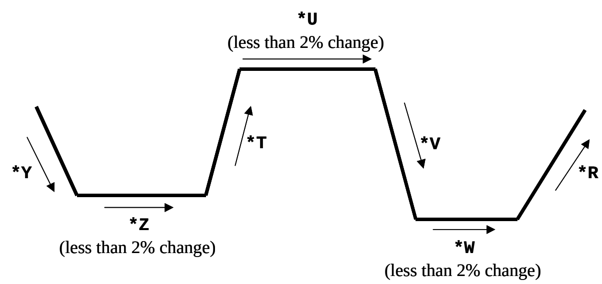

Figure 6: The relaxed double bottom pattern.

While the GSW algorithm is sufficient to handle examples listed so far, a minor extension is needed to handle the of Example 10, where inequalities have the form say $X\ op\ C{*}Y$. Here we can take advantage of the fact that the domain of Y is positive numbers (stock prices) and introduce a new variable $Z = X/Y$. Then we work with $Z\ op\ C$ instead of the original $X\ op\ C * Y$.

The runtime execution of SQL-TS is achieved via user-defined aggregates that are capable of applying arbitrary SQL statements on input streams [17].

7. EXPERIMENTAL RESULTS

In order to measure performance, we count the number of times that an element of input is tested against a pattern element. The speedups obtained range from the modest one obtained for the simple search pattern of Figure 4.2.1, to speedups of more than two orders of magnitude obtained on the complex patterns found in actual applications. For instance, a common search in stock market data analysis is for a double-bottom pattern, where the stock price has two consecutive local minima. Therefore, in our experiment we searched for "relaxed double-bottoms" in the recorded closing value of the DJIA (Dow Jones Industrial Average) index for the last 25 years. By a relaxed double bottom we mean a local maximum surrounded by two local minima, where we only consider the increases or decreases which are more than 2%. In other words, if the price moves less than 2%, we consider it as if it hasn't changed. (Figure 6).

Example 10 expresses the relaxed double bottom pattern in SQL-TS; `*Z`, `*U`, and `*W` represent the areas where changes are less than 2% and the curve is considered approximately flat (Figure 6). This query, optimized using the OPS algorithm, executes 93 faster than the naive execution on the DJIA's data for the last 25 years. Figure 7 shows there are 12 matches found in the input. The graph in the bottom of Figure 7 shows one of this patterns that occurred around June 1990. We ran several queries with complex search patterns, and measured speedups up to 800 times over naive search.

EXAMPLE 10. *Relaxed double bottom*

```
SELECT X.NEXT.date, X.NEXT.price,
       S.previous.date, S.previous.price
FROM djia
   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
```

8. FURTHER WORK & CONCLUSION

In this paper, we described a novel approach for querying complex sequential patterns and optimizing these queries. We are currently pursuing various improvements and extensions, on which we next present a very short summary, due to space limitations.

We have developed a method for calculating ϕ and θ for a more general class of predicates that includes predicates on intervals (open and closed intervals, single-dimensional and multidimensional ones) [13]. Our method transforms implication and satisfiability problems into set inclusion problems in the domain of intervals and their complements; we can then handle the search for patterns in a spatio-temporal database [13]. We have also extended the OPS algorithm to optimize patterns containing disjunctive conditions [13].

Clearly, it is possible to search the input stream in either the forward or the reverse direction. Therefore, we can optimize searches in both directions, and then select the better. We are currently seeking good heuristics for selecting the more effective of the two optimizations. For instance, a large average value for $shift$ and $next$ is a good indication of effective optimization. Specially a larger value of $shift$ has more effect on the speedup.

Finally, we are investigating the suitability of other pattern search algorithms for extensions similar to those we have used for KMP. Although there is evidence that KMP provides better performance on the average [18], than other algorithms, such as those by Karp&Rabin [7] and Boyer&Moore [10], could offer some advantage in special situations.

9. ACKNOWLEDGEMENTS

This work was partially supported by the National Science Foundation under grant: NSF–IIS 0070135.

The authors are grateful to the reviewers for the several improvements they suggested, and to Elias Koutsoupias for his advice.

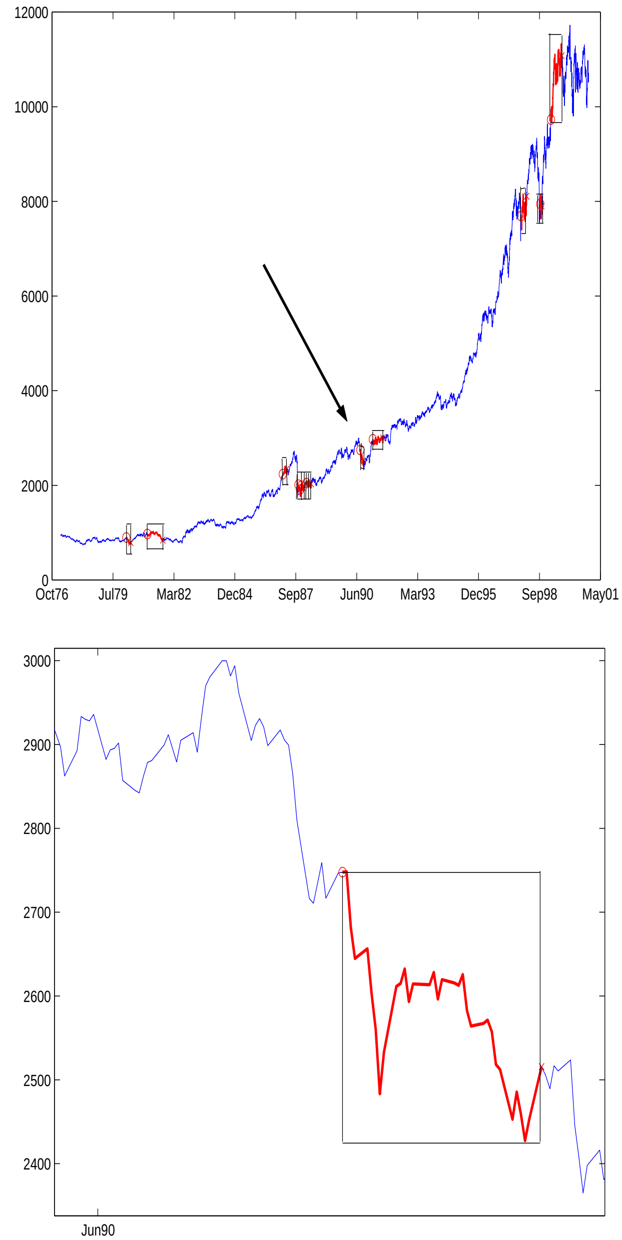

Figure 7: Doublebottoms found in the DJIA data are shown by boxes. The bottom picture is zoomed for the area pointed by arrow in the top picture and shows one of the matches.

10. REFERENCES

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *International Conference on Data Engineerin*, 1995.

[2] M. Berry and G. Linoff, *Data Mining Techniques: For Marketing, Sales, and Customer Support*. John Wiley, 1997.

[3] R.D. Edwards and J. Magee. *Technical Analysis of Stock Trends*. AMACOM, 1997.

[4] C. Faloutsos, M. Ranganathan, and Manolopoulos Y. Fast subsequence matching in time-series databases. In *Proc. Int. Conf. On Management of Data*, pages 419–429, 1994.

[5] S. Guo, W. Sun, and M. Weiss. On satisfiability, equivalence, and implication problems involving conjunctive queries in database systems. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):604–616, August 1996.

[6] Informix Software Inc. Managing time-series data in financial applications, 1998. White Paper.

[7] R. Karp and M. O. Rabin. Efficient Randomized Pattern Matching Algorithm. *IBM Journal of Research and Developement*, 31(2):249–260, March 1987.

[8] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, June 1977.

[9] E. Mesrobian et al., Extracting spatio-temporal patterns from geoscience datasets. In *IEEE Workshop on Visualization and Machine Vision*, 1994.

[10] J. S. Moore and R. S. Boyer. A Fast String Searching Algorithma. *Communications of ACM*, 20(10):762-772, 1977.

[11] I. Motakis and C. Zaniolo. Temporal aggregation in active databases. In *Int. Conf. on the Management of Data*, May 1997.

[12] R. Ramakrishnan et al., SRQL: sorted relational query language, SSDBM 1998: 84-95.

[13] R. Sadri, Optimization of Sequence Queries in Database Systems Ph.D. Thesis, UCLA, 2001.

[14] P. Seshadri. Predator: A resource for database research. *SIGMOD Record*, 27(1):16–20, 1998.

[15] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query processing. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 430–441, May 1994.

[16] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A model for sequence databases. In *ICDE*, pages 232–239, 1995.

[17] H. Wang and C. Zaniolo. Using SQL to Build New Aggregates and Extenders for Object-Relational Systems. In *Proceedings of 26th International Conference on Very Large Data Bases*, 2000.

[18] C. A. Wright, L. Cumberland and Y. Feng, A Performance Comparison Between Five String Pattern Matching Algorithms, Dec. 98 Tech.Report, `http://ocean.st.usm.edu~/cawright/pattern_matching.html`