Expressing and Optimizing Sequence Queries in Database Systems

REZA SADRI
Procom Technology Inc., Irvine, California
CARLO ZANIOLO
UCLA Computer Science Department, Los Angeles, California
AMIR ZARKESH
3Plus1 Technology, Inc., Saratoga, California
and
JAFAR ADIBI
Information Sciences Institute, USC, Marina del Rey, California

The need to search for complex and recurring patterns in database sequences is shared by many applications. In this paper, we investigate the design and optimization of a query language capable of expressing and supporting efficiently the search for complex sequential patterns in database systems. Thus, we first introduce SQL-TS, an extension of SQL to express these patterns, and then we study how to optimize the 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 interdependencies between the elements of a 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.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages—*query languages*; H.2.4 [**Database Management**]: Systems—*query processing*

General Terms: Algorithms, Theory, Languages

Additional Key Words and Phrases: Time series, sequences, query optimization, searching

1. INTRODUCTION

Many applications require processing and analyzing sequential data to detect pattern and trends of interest. Examples include the analysis of stock

This work was partially supported by the National Science Foundation under grant IIS-0070135.
Authors' addresses: R. Sadri, Procom Technology, Inc., 58 Discovery, Irvine, CA 92618; email: sadri@procom.com; C. Zaniolo, CS Dept., UCLA, Los Angeles, CA 90095; email: zaniolo@cs.ucla.edu; A. Zarkesh, 3Plus1 Technology, Inc., 18809 Cox Avenue, Suite 250, Saratoga, CA 95070; email: azarkesh@comcast.net; J. Adibi, ISI, USC, 4676 Admiralty Way, Suite 1001, Marina del Rey, CA 90292; email: adibi@isi.edu.

market prices [Edwards and Magee 1997], meteorological events [Mesrobian et al. 1994], and the identification of patterns of purchases by customers over time [Agrawal and Srikant 1995; Berry and Linoff 1997]. The patterns of interest range from very simple ones, such as finding three consecutive sunny days, to the more complex patterns used in data mining applications [Agrawal and Srikant 1995; Faloutsos et al. 1994; Informix Software 1998].

The importance of these applications have motivated work to extend database query languages with the ability of searching for and manipulating sequential patterns. Informix [Informix Software 1998] was the first among commercial DBMSs to provide special libraries for time-series, that they named datablades; these libraries consist of functions that can be called in SQL queries. While other database vendors were quick to embrace it, this procedural-extension approach lacks expressive power and amenability to query optimization. Indeed, while the individual datablade functions are highly optimized for their specific tasks, there is no optimization between these functions and the rest of the query.

To solve these problems, the SEQ and PREDATOR systems introduce a special sublanguage, called SEQUIN for queries on sequences [Seshadri et al. 1994, 1995; Seshadri 1998]. SEQUIN works on sequences in combination with SQL working on standard relations; query blocks from the two languages can be nested inside each other, with the help of directives for converting data between the blocks. SEQUIN's special algebra makes the optimization of sequence queries possible, but optimization between sequence queries and set queries is not supported; also its expressive power is still too limited for many application areas. To address these problems, SRQL [Ramakrishnan et al. 1998] augments relational algebra with a sequential model based on sorted relations. Thus sequences are expressed in the same framework as sets, enabling more efficient optimization of queries that involve both [Ramakrishnan et al. 1998]. SRQL also extends SQL with some constructs for querying sequences.

SQL/LPP is a system that adds time-series extensions to SQL [Perng and Parker 1999]. SQL/LPP models time-series as attributed queues (queues augmented with attributes that are used to hold aggregate values and are updated upon modifications to the queue). Each time-series is partitioned into segments that are stored in the database. The SQL/LPP optimizer uses pattern-length analysis to prune the search space and deduce properties of composite patterns from properties of the simple patterns. Here too, the pattern language is largely decoupled from SQL, bringing problems similar to those of SEQ. Moreover, SQL/LPP doesn't detect recursive patterns, and only supports a limited set of aggregate functions. While, it is possible to build more complex aggregates combining these basic functions, new aggregate functions cannot be introduced from scratch.

There has also been a significant amount of work on extending SQL triggers to detect composite events in Active Databases [Gehani et al. 1992; Gatziu and Dittrich 1993; Motakis and Zaniolo 1997]. The languages used in these systems support some of the key functions needed for sequence analysis, including a marriage of regular expressions with SQL, and temporal aggregates.

However, the implementation and optimization techniques needed to satisfy the special (update and transaction) requirements of active databases are not present in sequence queries, which therefore provide greater opportunities for query optimization, which are discussed next.

In this article, we explore optimization techniques inspired by string-search algorithms, since finding sequential patterns in databases is somewhat similar to finding phrases in text. The naive approach, which advances the search by one position and restart from the beginning of the pattern after each failure, has time complexity $O(m \times n)$, where m is the length of the text and n the length of the pattern. The Karp–Rabin algorithm [Karp and Rabin 1987] has a worst time complexity of $O(n \times m)$ and an expected running time of $O(n + m)$; the algorithm works by hashing the values of possible substrings of size m, and its efficiency depends on the alphabet size. The Boyer–Moore pattern matcher [Boyer and Moore 1977] works best when the pattern is long and the alphabet is large. The worst case performance of this pattern matcher is $O(n \times m)$, and its best case performance is $O(n/m)$. The algorithms discussed so far assume a finite alphabet size. The Knuth–Morris–Pratt (KMP) algorithm discussed next does not suffer from this limitation.

The KMP algorithm [Knuth et al. 1997] creates a prefix function from the pattern to define transition functions that expedite the search. The prefix function is built in $O(m)$ time, and the algorithm has a worst case time complexity of $O(n+m)$, independent from the alphabet size. Exhaustive experiments [Wright et al. 1998] show that, in general, KMP has the best performance. Because of its good performance, and its independence from the alphabet size, KMP provides a natural basis for dealing with the more general problem of optimizing database queries on sequences. This is a major generalization that presents difficult challenges: rather than searching for strings of letters (usually from a finite alphabet), we have now to search for sequences of structured tuples qualified by arbitrary expressions of propositional predicates involving arithmetic and aggregates.

The article is organized as follows. In the next section, we introduce the SQL-TS query language, and in Section 3 we introduce the query optimization problem as an extension of the text searching problem. Our new algorithm for query optimization is introduced in Section 4, and then extended to handle stars and aggregates in Section 6. The performance of the new approach is studied in Section 6. Generalizations of the algorithm for disjunctive patterns are described in Section 7.

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), price Integer, date Date)
```

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
...	...	...

Fig. 1. Effects of `SEQUENCE BY` and `CLUSTER BY` on data.

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 2.1:

Example 2.1. 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 (see appendix A for the specification of the syntax of these extensions).

—A `CLUSTER BY` clause specifies that data for the different stocks are processed separately (i.e., as if they arrived in separate data streams.) The semantics of this construct is basically same as the `PARTITIONED BY` construct used in SQL:1999 windows [Zemke et al. 1999; Alur et al. 2002]. This semantics has also been in recently proposed SQL extensions for data streams [Babcock et al. 2002].

—A `SEQUENCE BY date` clause specifies 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).
The `SEQUENCE BY` attributes(s) is similar to the `ORDERED BY` construct used in SQL:1999 [Zemke et al. 1999; Alur et al. 2002]. Similar constructs were also used in SRQL, which supports `GROUP BY` and `SEQUENCE BY` clauses [Ramakrishnan et al. 1998].

—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.

For a second example, consider the log of the web pages clicked by a user during a session:

```
Sessions(SessNo, ClickTime, PageNo, PageType)
```

A user entering the home page of a given site starts a new session that consists of a sequence of pages clicked; for each session number, `SessNo`, the log shows the sequence of pages visited—where a page is described by its timestamp, `ClickTime`, number, `PageNo` and type `PageType` (e.g., a content page, a product description page, or a page used to purchase the item).

The ideal scenario for advertisers is when users (i) see the advertisement page for some item in a content page, (ii) jump to the product-description page with details on the item and its price, and finally (iii) click the 'purchase this item' page. This advertisers' dream pattern can expressed by the following SQL-TS query, where 'a', 'd', and 'p', respectively, denote an ad page, an item description page, and a purchase page:

Example 2.2. Using the `FROM` clause to define patterns

```
SELECT Y.PageNo, Z.ClickTime
FROM Sessions
    CLUSTER BY  SessNO
    SEQUENCE BY  ClickTime
    AS (X, Y, Z)
WHERE  X.PageType='a'
  AND  Y.PageType='d'
  AND  Z.PageType='p'
```

Thus, the `CLUSTER BY` clause specifies that data for each `SessNO` are processed as separate streams; instead, the `SEQUENCE BY` clause specifies that the tuples for each `SessNO` are ordered by ascending `clickTime`. Finally, the pattern `AS (X, Y, Z)` specifies that, for each `SessNO`, we seek a sequence of the three tuples `X, Y, Z` (with no intervening tuple allowed) that satisfy the conditions stated in the `WHERE` clause.

Observe that in the `SELECT` clause, we return information from both the `Y` tuple and the `Z` tuple. This information is returned immediately, as soon as the pattern is recognized; thus it generates another stream that can be cascaded into another SQL-TS statement for processing.

The next example illustrates how SQL-TS benefits from its ability of using standard SQL queries in combination with queries on sequences. Assume that we have a stream containing the bids of ongoing auctions, as follows:

`auctn_id :`	*id for specific item auctioned*
`amount :`	*amount of bid*
`time :`	*timestamp*

Say that our objective is to purchase the auctioned item for a low price. Then, we wait till the last 15 minutes before the closing, and we place an offer as soon as the stream of bids is converging toward a certain price. We detect convergence by a succession of three bids that raise the last bid by less than 2%. Such convergence conditions can be expressed as follows:

```
SELECT T.auctn_id, T.timestamp, T.amount
FROM bids CLUSTER BY auctn_id
          SEQUENCE BY time
          AS (X,Y,Z,T)
WHERE   Y.amount < 1.02 * X.amount
    AND Y.amount > .98 * Z.amount
    AND T.amount < 1.02 * Z.amount
```

This query specifies that the `Y.amount` must be above `X.amount` by 2% or less, and the same condition must hold between `Z` and `Y`. To assure that we are within 15 minutes from closing, we use a standard SQL query on the table where the auctions are described:

```
auction(auctn_id, item_id, min_bid, deadline,...)
```

Our query becomes:

Example 2.3. Three successive bids with a 2% range in the 15 minutes before closing

```
SELECT T.auctn_id, T.timestamp, T.amount
FROM  auction AS A,
      bids CLUSTER BY auctn_id
           SEQUENCE BY time
           AS (X,Y,Z,T)
WHERE   A.auctn_id = T.auctn_id
    AND T.time + 15 Minute < A.deadline
    AND Y.amount < 1.02 * X.amount
    AND Y.amount > .98  * Z.amount
    AND T.amount < 1.02 * Z.amount
```

The `WHERE` conditions of this query specify various predicates that must be satisfied by the attributes of four tuples `X`, `Y`, `Z`, `T` in a sequence. The evaluation of the applicable predicates on these four variables, however, is not delayed until all four tuples are read; instead each predicate is evaluated as soon all its variables in the predicate are known—that is, as soon as the *predicate becomes fully instantiated*.

For instance, the predicate `Y.amount` $<$ `1.02`$*$`X.amount` is fully instantiated at `Y`, since we already know all the values in `X` when the tuple `Y` is read. However, the same predicate is not fully instantiated at `X`, since, when we read `X`, we do not yet know the values in `Y`. Therefore, when matching the input to the pattern in the previous example, the first input tuple is read and assigned to `X` without any condition checked; but, as soon as the next input tuple is assigned to `Y`, we immediately check whether `Y.amount` $<$ `1.02`$*$`X.amount` is satisfied. If this check fails, we restart from the beginning, otherwise we proceed and read the next tuple for the attribute values of Z.

In SQL-TS, input tuples are viewed as containing the additional field previous that refers to the previous tuple in the sequence. For instance, the condition Y.amount $<$ 1.02 $*$ X.amount could have also been written as Y.amount $<$ 1.02 $*$ Y.previous.amount. (The SQL3 syntax Y.previous $\rightarrow$ amount is also supported.)

2.1 Repeating Patterns and Aggregates

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

Example 2.4. 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
```

Here the star construct $*$Y is used to specify a sequence of *one or more* Y's of decreasing price, as per the condition Y.price < Y.previous.price. In general, a star such as $*$Y denotes a maximal sequence of *one or more* (not zero or more!) tuples that satisfy all the applicable conditions. Thus, a star pattern such as $*$Y fails only when the predicates that become fully instantiated at Y fail on the first input. However, if such predicates succeed on the first $n \geq 1$ tuples and fail on tuple $n + 1$, then $*$Y succeed and completes on the nth tuple, and the $n + 1$ tuple is tested against the element in the pattern immediately following $*$Y (i.e., Z in Example 2.4).

Thus, in our Example 2.4, we begin with an arbitrary tuple X, and then, if the next tuple Y, satisfies the condition Y.price $<$ Y.previous.price $=$ X.Price we begin $*$Y. Then, we exit the star on the last decreasing price. Thus, Z is the first tuple in the sequence where the price has not decreased. Thus, Z.previous.price $<$ 0.5 $*$ X.price can now be used to detect a down sequence causing the stock to lose half of its value. Constructs similar to the star have been tested very effective in previously query languages [Motakis and Zaniolo 1997], and their semantics can be formalized using recursive Datalog programs [Sadri 2001].

Aggregates can be used in conjunction with stars. For instance, to determine the number of pages the user has visited before clicking a product description page (denoted by 'd'), we simply write:

Example 2.5. Number of pages visited before the product description page is clicked, provided that this count is below 20

```
SELECT SessNo, count(*A)
FROM Sessions
     CLUSTER BY  SessNO
     SEQUENCE BY  ClickTime
     AS (*A, B)
WHERE A.PageType <> ‘d’
  AND B.PageType = ‘d’
  AND count(*A) < 20
```

Thus, `*A` identifies a maximal sequence of clicks to pages other than ‘product’ pages. Then, `count(*A)` tallies up those pages and, after checking that the count is less than 20, returns `SessNo` and the associated count to the user. The maximality of stars construct is important to avoid ambiguity and the possible explosion of matches. For instance, if we were to change the first condition in the query of our Example 2.5 to, say, `A.PageType = ‘d’`, we obtain a query that is never satisfied, since the star consumes every ’d’ value, leaving none to satisfy the next condition: `AND B.PageType = ‘d’`. For instance, say that we specify a pattern `(*X, *Y)` and the following conditions in the where clause: `X<=5 AND Y>=5`. Then in the sequence 4, 5, 5, 7, `*X` will match the first 3 values, and only the fourth value (i.e., 7) will be left for `*Y)`. A user who wants to match `*X` to the first value and the next three values to `*Y`, will have to change the conditions to `X<5 AND Y>=5`. SQL-TS supports a rich set of aggregates, as needed for time series analysis [Berry and Linoff 1997]; aggregates supported includes rollups, running aggregates, moving-window aggregates, online aggregates, and user-defined aggregates inherited from the AXL/ATLaS system [Wang and Zaniolo 2000]. Aggregates can only be applied to sequences defined by stars, and come in two very distinct flavors:

(1) final aggregates applicable only after the star computation has completed, and
(2) continuous aggregates that apply during the star computation.

For instance, `count(*A)` in Example 2.5 is a final aggregate: a sequence of pages is accepted, until a ‘p’ page terminates the sequence. At that point, the condition `count(*A) < 20` is evaluated, and if satisfied the sequence is accepted and `SessNo` and `count(*A)` for that session are returned, otherwise the sequence is rejected.

Example 2.6 instead illustrates the use of continuous aggregates—that is, those that return the current value of the aggregates during the computation, as per online aggregates [Hellerstein et al. 1997]. For instance, the query in Example 2.6 uses continuous aggregates to detect sessions (identified by their `SessNo`) in which users have accumulated too many clicks, or spent too much time, without purchasing anything. The aggregate `ccount` is the online version of count, that is, a continuous count that returns a new value for each new input. Thus, the condition `ccount(X) < 100` is satisfied for the first 99 elements in the sequence and, upon failing on the 100th element, it brings the star sequence to completion. In general, continuous aggregates can be returned at various points during the computation of the sequence, as online aggregates do [Hellerstein et al. 1997]; thus, they can also be used in the conditions that determine whether the current tuple must be added to the star sequence being recognized.

The two different kinds of aggregates are syntactically distinguished by the fact that, the argument of a final aggregate is prefixed by the star; while there is no star in the argument of continuous aggregates.

Another continuous aggregate used in the next query is `first(X)`; this is a built-in aggregate that always returns the first value passed to it (thus, in Example 2.6, memorizes the first value of `ClickTime` value in the sequence `*X`.)

Example 2.6. Excessive clicks or time without a purchase

```
SELECT  Y.SessNo
FROM Sessions
     CLUSTER BY  SessNO
     SEQUENCE BY  ClickTime
     AS (*X, Y)
WHERE X.PageType<>‘p’
 AND  ccount(X) < 100
 AND  first(X.ClickTime) + 20 Minute >
      X.ClickTime AND  Y.PageType<>‘p’
```

Therefore, the recognition of `*X` begins and continues while (i) there is no purchase, (ii) the length of `*X` is less than 100 clicks, and (iii) the time elapsed is less than 20 minutes. Once any of these conditions fails, the sequence `*X` reaches completion. At the next click (assuming that this is not a ‘p’ page) `SessNo` is returned. (This could, e.g., trigger a time-out message to the remote users, requesting them to login again to continue the session.) Therefore, we use the `WHERE` clause to specify conditions on both the values of attributes and those of aggregates. This is a simplification of traditional SQL (that would instead require `HAVING` for conditions on aggregates). This simplification is very beneficial for the users, and it has been adopted in more recent query languages such as XQuery [Boag et al. 2003].

The simplification is made possible by the lack of ambiguity associated with the sequential processing of sequences of tuples. The processing is as follows: for each new tuple (i) the current values of attributes and continuous aggregates (i.e., those without the star, such as `ccount(X)`) are evaluated and all the applicable conditions in the `WHERE` clause are tested, and (ii) if said conditions evaluate to true, then the computation of the star continues with the next tuple. If the current tuple fails to satisfy said conditions clause, then the final aggregates such as `count(*X)` are computed and their values are used to test the applicable conditions in the where clause. If these conditions are satisfied, then the computation continues with the next tuple and the next element in the pattern; otherwise the current input fails, and the search is moved to a later input.

In general, therefore, we treat conditions on starred aggregates like conditions in the `HAVING` clause of standard SQL. Thus, for Example 2.5, the statement `WHERE count(*A) < 20` is treated like `HAVING count(A) < 20`.

Finally, the meaning of an aggregate such as `avg(*A)` would become undefined if `*A` were to contain zero or more elements (instead of one or more elements). Therefore, SQL-TS design attempts to achieves both users’ convenience and rigorous semantics. A formal logic-based semantics for the language is presented in Sadri [2001].

2.2 User-Controllable Options

The system provides the user with optional constructs to control the input and the output. The user can specify whether the input is sorted in ascending or descending order, and whether null values will be listed at the beginning or at the end, using the statements described in the Appendix. When these specifications are omitted, the system uses ascending-order and nulls-at-the-end as defaults.

For the output, the user can write `SELECT ALL`, or `SELECT DISJOINT`, to specify whetehr that overlapping subsequence are, or are not, acceptable. Thus, `SELECT DISJOINT` specifies that when a sequence starting at j and ending at $k > j$ is found to satisfy the query, the input tuples between j and k are ignored, and the search resumes from point $k + 1$. This is also the policy followed by the system when no explicit specification is given. Instead, with `SELECT ALL` success has no effect on successive matches. The actual syntax for these constructs is specified in the Appendix.

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 query of Example 2.2, which searches for the sequence of three particular constant values: the text searching algorithms by Knuth, Morris and Pratt (KMP), discussed next, provides a solution of proven optimality for this query [Knuth et al. 1997; Wright et al. 1998].

3.1 Searching for Simple Text Strings

The KMP algorithm takes a sequence pattern of length m, $P = p_1 \cdots p_m$, and a text sequence of length n, $T = t_1 \cdots t_n$, and finds all occurrences of P in T. Using an example from Knuth et al. [1997], 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 ith element in the text is compared with the jth 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							
				$\Uparrow$													

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

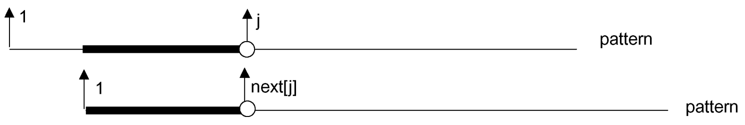

Fig. 2. The meaning of *next*(j).

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 $p_1 \cdots p_4 = p_4 \cdots p_7$ and $p_4 \ldots p_7 = t_8 \cdots t_{11}$, $p_1 \neq p_2$, 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 Knuth et al. [1997], 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 computed 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., $p_1 = p_{j-k+1} \wedge \cdots \wedge p_{k-1} = p_{j-1}$).
(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 $j - k + 1$).

For instance, for the example at hand, we find the following array: *next* $=$ $[0, 0, 0, 0, 0, 0, 0, 4, 0, 0]$. The definition of *next* 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 i, 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.)

Algorithm 3.1. The KMP Algorithm

```
$j = 1$;  $i = 1$;
while $j \le m \wedge i \le n$ do {
  while $j > 0 \wedge t_i \ne p_j$ 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 Knuth et al. [1997]. 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 [Knuth et al. 1997]. When success occurs, the input text $t_{i-m+1} \cdots t_i$ matches the pattern.

The KMP algorithm is only applicable when the qualifications in the query are equalities with constants such as those of Example 2.2. Therefore, in this article, 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 3.2. For IBM stock prices, find all instances where we have 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
```

4. GENERAL PREDICATES

The original KMP algorithm can be used to optimize simple queries, such as that of Example 2.2, in which conditions in the WHERE clause are equality predicates as follows (t denotes a generic 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 also need to support:

(1) *General Predicates.* In particular we need to support systems of equalities and inequalities such as those of Example 3.2, 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 Examples 2.4–2.6, we need to optimize searches involving recurring patterns expressed by the star.

(3) *Aggregates.* Patterns can be specified using a variety of aggregates, including windows-based, temporal, and user-defined aggregates.

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 and aggregate conditions on such repeating patterns.

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

(1) All conditions for elements 1 through $j - 1$ in the search pattern were satisfied by the corresponding items in the input sequence, and

(2) The condition for the jth 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 relationships 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 is never shifted in the text (except for the case where *next*(j) $= 0$ and the pattern is 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 mxm, 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 θ_{jk} and ϕ_{jk} as follows:

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

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

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 cases, we have removed the ambiguity. Logic matrices θ and ϕ contain all the possible pairwise logical relations between pattern elements. For instance, Example 4.1 shows the computation of the matrices for Example 3.2.

Example 4.1. Computing the matrices θ and ϕ for Example 3.2

$p_2 \Rightarrow p_1$	therefore	$\theta_{21} = 1$
$p_3 \Rightarrow \neg p_1$	therefore	$\theta_{31} = 0$
$p_3 \Rightarrow \neg p_2$	therefore	$\theta_{32} = 0$
$p_4 \Rightarrow \neg p_2$	therefore	$\theta_{42} = 0$
$p_4 \Rightarrow \neg p_1$	therefore	$\theta_{41} = 0$
$\neg p_4 \Rightarrow \neg p_3$	therefore	$\phi_{43} = 0$

Therefore, we have

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

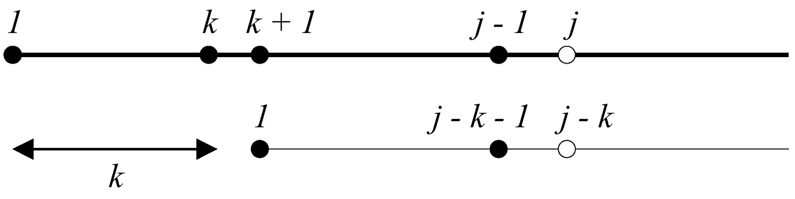

Fig. 3. Shifting the pattern k positions to the right.

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

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:

Example 4.2. Computing the matrix S for Example 4.1

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

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

We can now compute *shift*(j), which is the least shift to the right for which the overlapping subpatterns do not contradict each other (Figure 4). Thus, *shift*(j) is the column number for the leftmost nonzero 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

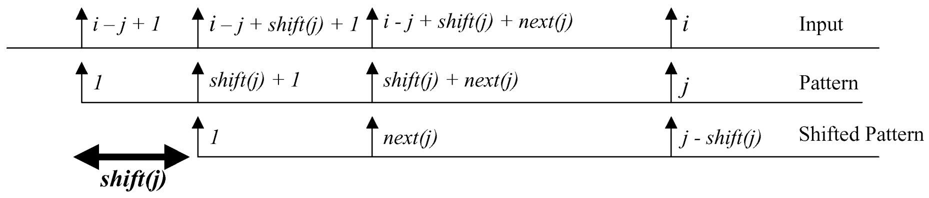

Fig. 4. Next and Shift definitions for OPS.

text. More formally:

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

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 cases. 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) $next(j) = \mathbf{min}(\{t \mid 1 \leq t < j - shift(j) \wedge \theta_{shift(j)+t,t} = U\} \cup \{j - shift(j) | \phi_{j,j-shift(j)} = U\})$

For the example at hand, we have:

Example 4.3. Compute *shift* and *next* for Example 4.1

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

The calculation of arrays *shift* and *next* is done as part of query compilation. This is discussed in Section 4.3.

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

Algorithm 4.4. 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 $t_{i-m+1} \ldots t_i$ satisfies 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 ith element in the input.

—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 3.2 to search the following sequence:

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

Figure 5 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.

4.3 Calculating θ and ϕ

As described in the previous section, the OPS algorithm is based on the two arrays *shift* and *next*, which are computed from logic arrays θ and ϕ. Here we discuss efficient algorithms for computing these logic arrays.

Elements of ϕ and θ are calculated in accordance with the semantics of the pattern elements. Satisfiability and implication results in databases [Guo et al. 1996a; Ullman 1989; Klug 1988; Rosenkrantz and Hunt 1970; Sun and Yu 1994; Sun et al. 1989] are relevant to the computation of θ and ϕ for a class of patterns that involve inequalities in a totally ordered domain (such as real numbers). Ullman [1989] has given an algorithm for solving the implication problem between two queries S and T. Ullman's algorithm works for queries which are conjunctions of terms of the form $X\ op\ Y$, where $op \in \{<, \leq, =, \neq, \geq, >\}$, and has complexity of $O(|S|^3 + |T|)$, where $|S|$ and $|T|$, respectively, denote the number of inequalities in S and T.

Klug [1988] has studied the implication problem in a broader range of queries that are conjunction of terms of the form $X\ op\ C$ and $X\ op\ Y$. Rosenkrantz and

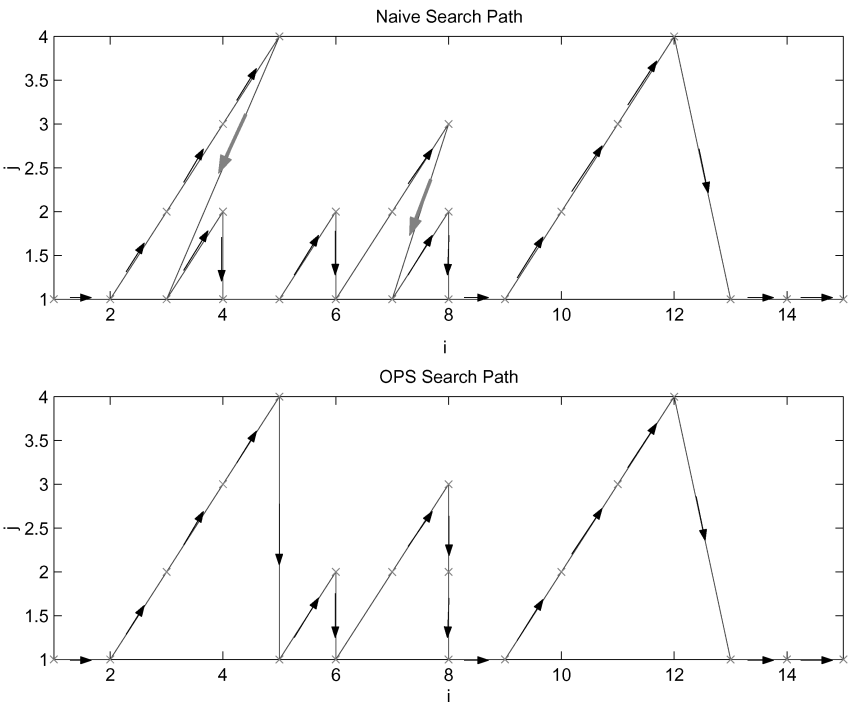

Fig. 5. Comparison between path curve of the naive search (top chart) and OPS (bottom chart).

Hunt [1970] provided an algorithm complexity of complexity $|S|^3$ for solving satisfiability problem; the expression S to be tested for satisfiability is the conjunction of terms of the form X *op* C, X *op* Y, and X *op* $Y + C$.

In our implementation, we compute the matrices ϕ and θ using the algorithms by Guo, Sun and Weiss (GSW) [Guo et al. 1996a] discussed next.

4.4 The GSW Algorithm

The GSW algorithm computes implication and satisfiability of conjunctions of 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* $\in \{=, \neq, \leq, \geq, <, >\}$. Implication and satisfiability are, respectively, used to infer the 1 entries and the 0 entries of our θ and ϕ matrices. The complexity of GSW algorithm is $O(|S| \times n^2 + |T|)$ for testing implication (for the 1 entries in our matrices) and $O(|S| + n^3)$ for testing satisfiability (for the 0 entries); n is the number of variables in S and $|S|$, and $|T|$ denote the number of inequalities in S and T. Given the limited number of variables and inequalities used in queries, these compilation costs are quite reasonable. GSW starts with applying the following transformations:

(1) $(X \geq Y + C) \equiv (Y \leq X - C)$
(2) $(X < Y + C) \equiv (X \leq Y + C) \wedge (X \neq Y + C)$
(3) $(X > Y + C) \equiv (Y \leq X - C) \wedge (X \neq Y + C)$
(4) $(X = Y + C) \equiv (Y \leq X - C) \wedge (X \leq Y + C)$

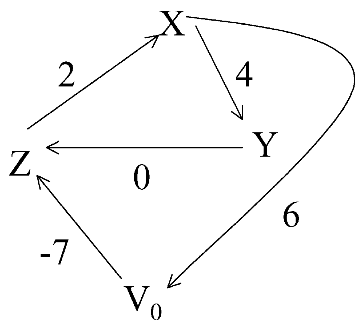

Fig. 6. Directed weighted graph for determining the satisfiability of a set of inequalities.

(5) $(X < C) \equiv (X \leq C) \wedge (X \neq C)$
(6) $(X > C) \equiv (X \geq C) \wedge (X \neq C)$
(7) $(X = C) \equiv (X \leq C) \wedge (X \geq C)$.

After these transformations, for all the inequalities of the form $X\ op\ Y + C$, we have $op \in \{\leq, \neq\}$, and for all the inequalities of the form $X\ op\ C$, $op \in \{\leq, \neq, \geq\}$ ($X\ op\ Y$ is a special form of $X\ op\ Y + C$ where $C = 0$).

4.5 Satisfiability

For determining the satisfiability of a conjunctive query S, a directed weighted graph $G_s = (V_s, E_s)$ is built where V_s is the set of variables in S, and there is a directed edge from X to Y with weight C in E_s, if and only if $(X < Y + C) \in S$. Inequalities of the form $(X < C)$ are transformed to the form $(X < V_0 + C)$ by introducing dummy variable V_0. Thus, the following results are proven in Guo et al. [1996a]: If there is a *negative weighted cycle*—a cycle that sum of the weights of its edges is negative, then S is unsatisfiable. If all the cycles are *positive weighted*, then S is satisfiable. For the case that there are *zero weighted cycles*, the necessary and sufficient condition for satisfiability is that for any two variables X and Y on the same cycle, if the path from X to Y has a cost C, then $(X \neq Y + C) \in S$. As shown in Guo et al. [1996a], this algorithm has the time complexity of $O(|S| + n^3)$ where $|S|$ is the number of inequalities in S and n is the number of variables (size of V_s). The following example clarifies how the algorithm works:

Example 4.5. Assume that we want to find out if θ_{jk} is zero or not where the two pattern elements p_j and p_k are as follows:

$$\begin{aligned} p_j &= X < Y + 4 \wedge Y < Z \\ p_k &= Z < X + 2 \wedge X < 6 \wedge Z > 7 \end{aligned}$$

To see if $p_j \wedge p_k$ is satisfiable or not, we first build a graph for $p_j \wedge p_k$ as in Figure 6.

There are two cycles in the graph. Cycle *XYZX*, has weight of 6 and cycle *XV_0ZX* has weight of 1. Since there are no negative weighted cycles, $p_j \wedge p_k$ is satisfiable and value of θ_{jk} is not zero.

4.6 Implication

The implication problem takes two queries S and T and determines if S implies T. S and T are assumed to be conjunctions of inequalities of the form $X\ op\ Y + C$. For the inequalities of type $X\ op\ C$, a dummy variable V_0 is defined that can take only value of zero and the inequality is transformed to $X\ op\ V_0 + C$. As proven in Guo et al. [1996a], the application of this transformation does not change the answer to the implication problem. The algorithm starts by introducing the closure of S, that is, a complete set that contains all the inequalities implied by S. Then, T is implied by S Iff T is a subset of the closure of S. The notion of *modulo closure* of S, denoted $S_{closure}$, is then introduced to address the problem that the number of inequalities implied by S could be boundless. $S_{closure}$ contains only non redundant inequalities that belong to the closure of S. For example if $Y < X + C_1$ is in the closure of S, then for every $C_2 > C_1$, the inequality $Y < X + C_2$ is redundant. $S_{closure}$ can be computed by applying the following set of axioms to S [Guo et al. 1996a]:

A1. $X \leq X + 0$;

A2. $X \neq Y + C$ implies $Y \neq X - C$ where Y and X are distinct variables;

A3. $X \leq Y + C$ and $Y \leq V + C'$ implies $X \leq V + C + C'$;

A4. $X \leq W + C_1$, $W \leq Y + C_2$, $X \leq Z + C_3$, $Z \leq Y + C_4$, $W \neq C + Z$, and $C = C_3 - C_1 = C_2 - C_4$ imply $X \neq Y + C_1 + C_2$ where X and Y are distinct variables. Also Z and W are distinct variables.

As proven in Guo et al. [1996a], the size of $S_{closure}$ is finite, and calculating it has a time complexity of $O(|S| \times n^2)$. Furthermore, we have the following property [Guo et al. 1996a]:

PROPOSITION 4.6. *S implies T iff S is unsatisfiable or the following two properties hold:*

(1) *for every* $(X \leq Y + C) \in T$, *there exist* $(X \leq Y + C_0) \in S_{closure}$ *such that* $C_0 < C$, *and*

(2) *for every* $(X \neq Y + C) \in T$, *either*
—$(X \neq Y + C) \in S_{closure}$, *or*
—*there exist* $(X \leq Y + C_1) \in S_{closure}$ *such that* $C_1 < C$, *or*
—*there exist* $(Y \leq X + C_2) \in S_{closure}$ *such that* $C_2 < -C$.

This step takes $O(|T|)$ [Guo et al. 1996a]; therefore, the complexity of whole algorithm is $O(|S| \times n^2 + |T|)$.

While the GSW algorithm is sufficient to handle the examples listed so far, a minor extension is needed to handle the next query—Example 6.1. In this query, inequalities have the form $X\ op\ C * Y$. Then, we introduce a new variable $Z = X/Y$ and use $Z\ op\ C$, given that the domain of Y is positive numbers (stock prices).

In a later work, Guo et al. [1996b] found tighter bounds for these problems when the domain of variables are assumed to be the real numbers. They also showed that the satisfaction and implication problems become NP-hard when the problems must be solved in the domain of integers. However, if the inequality predicate is not allowed, the problems are polynomially bound in the integer domain as well.

5. PATTERNS WITH STARS AND AGGREGATES

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 if p_j is

$$\texttt{t}_\texttt{i}\texttt{.price} < \texttt{t}_{\texttt{i}-\texttt{1}}\texttt{.price}$$

then $*p_j$ matches sequences of records with decreasing prices.

The calculation of 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 *next* and *shift* must be generalized for star patterns as described next.

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. Take the following SQL-TS example:

Example 5.1. Find patterns consisting of a period of rising prices, followed by a period of falling prices, followed 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 star predicates that must be satisfied are as follows:

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

5.1 Run Time Support for Stars

A counter must be used for each element in the pattern. Let us represent the counter for the jth element of the pattern by $count_j$.

For instance, say that the previous query is applied to an input stream with the following sequence for $t.price$:

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

Then after matching the query pattern with the input, the counters contain the following values:

$\text{count}_1 = 4$	since the first four elements satisfy p_1
$\text{count}_2 = 9$	since the following five elements satisfy p_2
$\text{count}_3 = 11$	since two elements after that satisfy p_3.

We update and use these counters at run time. Then, to support star patterns, the OPS algorithm is modified as follows:

Algorithm 5.2. The OPS Algorithm for Patterns with Stars:

If the current input element satisfies the pattern then move to the next input, and

(1) if the current pattern element is not a star element then move to the next one, otherwise;
(2) update the current count.

Otherwise (i.e., when the current input element does not satisfy the pattern):

(1) If this is a star element, whose predicate has already been satisfied by the previous input element, move to the next pattern element and the next input.
(2) If this is not a star element, or is a star predicate 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:

$$i := i - \text{count}(j-1) + \text{count}(\mathit{shift}(j) + \mathit{next}(j) - 1).$$

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

5.2 Finding *next* and *shift* for the Star Case

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

$$\begin{tikzcd}
\theta_{21} \arrow[d] \arrow[dr] & & \\
\theta_{31} \arrow[r] \arrow[d] \arrow[dr] & \theta_{32} \arrow[d] \arrow[dr] & \\
\theta_{41} \arrow[r] \arrow[d] \arrow[dr] & \theta_{42} \arrow[r] \arrow[d] \arrow[dr] & \theta_{43} \\
{} & {} & {}
\end{tikzcd}.$$

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_{kj} where neither p_k nor p_j 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 $\theta_{kj} \to \theta_{k+1,j+1}$. However, if p_j is not as 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 $\theta_{kj} \to \theta_{k+1,j}$. In general, it is clear 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 $\theta_{kj} = 0$ entry must removed with all its incoming and departing arcs: we only retain entries that are either 1 or U.

Considering all possible situations, and assuming that all the neighbors are nonzero entries, we conclude that only the following transitions are needed when building the graph:

(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,

$$\begin{tikzcd} U \arrow[r] \arrow[d] \arrow[dr] & \theta_{j,k+1} \\ \theta_{j+1,k} & \theta_{j+1,k+1} \end{tikzcd} .$$

(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 $\theta_{j+1,k}$. Pictorially,

$$\begin{tikzcd} 1 \arrow[d] \arrow[dr] & \theta_{j,k+1} \\ \theta_{j+1,k} & \theta_{j+1,k+1} \end{tikzcd} .$$

Observe that there is no arc to $\theta_{j,k+1}$. This is because $\theta_{j,k} = 1$, and, therefore, all input tuples that satisfy p_j must also satisfy p_k.

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

$$\begin{tikzcd} \theta_{jk} \arrow[dr] & \theta_{j,k+1} \\ \theta_{j+1,k} & \theta_{j+1,k+1} \end{tikzcd} .$$

(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 $\theta_{j,k+1}$,

$$\begin{tikzcd} \theta_{jk} \arrow[r] \arrow[dr] & \theta_{j,k+1} \\ \theta_{j+1,k} & \theta_{j+1,k+1} \end{tikzcd} .$$

(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:

$$\begin{tikzcd} \theta_{jk} \arrow[d] \arrow[dr] & \theta_{j,k+1} \\ \theta_{j+1,k} & \theta_{j+1,k+1} \end{tikzcd} .$$

These rules assume that the end nodes of the arcs have value U or 1; but when such nodes have value 0, the incoming arcs will be dropped.

The directed graph produced by this construction will be called the *Implication Graph* for pattern sequence P, and is denoted as G_P. For each value of j, this graph must be further modified with entries from ϕ to account for the fact that 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 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 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 an star, then there will be only an arc from $\theta_{j-1,k}$ to row j that goes to ϕ_{jk}. Furthermore, all the original G_P entries in rows up to and including $j - 1$ remain unchanged, and so are 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).

For instance, in Example 5.3 below, we want to find occurrences of the following pattern in IBM's stock price: a period of increasing prices leading to a price between 30 and 40, followed by a period of decreasing price, followed by another period of increasing price leading to a price between 35 and 40, followed by a decreasing period leading to a price below 30. The query written in SQL-TS is:

Example 5.3. Looking for an $\mathcal{M}$-shaped pattern with specific high & low points

```
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:

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

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 can connect the elements of θ (after excluding the main diagonal) as follows:

$$G_P = \left[\begin{tikzcd} - & & & & & & \\ U \arrow[dr] & - & & & & & \\ 0 & U & - & & & & \\ 1 \arrow[d] \arrow[dr] & U \arrow[dr] & 0 & - & & & \\ U \arrow[dr] & 1 \arrow[dr] & U \arrow[d] & U \arrow[dr] & - & & \\ 0 & U \arrow[r] \arrow[dr] & 1 \arrow[d] \arrow[dr] & 0 & U \arrow[dr] & - & \\ U & 0 & U & U & 0 & U & - \end{tikzcd}\right].$$

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.

Then, we have the following graph:

$$G_P^6 = \left[\begin{tikzcd}[row sep=small, column sep=small]
- & & & & & \\
U \arrow[dr] & - & & & & \\
0 & U & - & & & \\
1 \arrow[d] \arrow[dr] & U \arrow[dr] & 0 & - & & \\
U \arrow[d] \arrow[dr] & 1 & U \arrow[dr] & U \arrow[d] \arrow[dr] & - & \\
U & U & 0 & U & U & -
\end{tikzcd}\right].$$

Consider now the node θ_{41} in this graph. Observe that there are several paths consisting of either 1 nodes or U nodes that take us to nodes in the last row of the matrix. Therefore, the input shifted by 4 can succeeds 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 is not path 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*$(6) = 3$.

Computation of shift(j) *and next*(j) *from* G_P^j. We compute *shift*(j) from the set of nodes in the first column of G_P^j that are sources of paths leading to the last row of G_P^j, as follows:

Definition 5.4. For a given pattern P, we define the set:

$$N_P^j = \{ n \mid \exists\, m \text{ such that } G_P^j \text{ contains a path from } \theta_{n,1} \text{ to } \phi_{j,m}\}.$$

Then,

—If the set N is not empty, *shift*(j) is one less than the least value in I: *shift*$(j) =$ *min*$(I) - 1$,

—otherwise (i.e., when the set N is empty) we set:

if $\phi_{j1} \neq 0$, then *shift*$(j) = j - 1$, else *shift*$(j) = j$.

Now we can define *next*. Note that there might be more than one path found in the definition of *shift*, but *next* must return a unique value to be used in restarting the search. 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, we set *next*$(j) = 1$. Otherwise, we move to the unique successor of this deterministic node and repeat the test. When the first nondeterministic node is found in this 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 nonzero path from node θ_{41} to node ϕ_{61}, thus *shift*$(6) = 3$. We now consider $\theta_{41} = 1$ and see that this is not a deterministic node, since there is more than one arc leaving the node. Thus, we conclude that *next*$(6) = 1$.

Complexity of Calculating Next and Shift. The G_P graph built in the last section has at most $m(m-1)/2$ nodes and each node has a constant number of (maximum 3) outgoing edges. Thus at the worst case, traversing the graph in depth first search manner for finding *shift*(j) will have the complexity of $O(j^2)$, and the worst-case complexity for finding all the *shift* values will be $O(m^3)$. Note that the first $j-2$ levels of graph G_P^{j-1} are the same as the first $j-2$ levels of graph G_P^j, therefore we can use the results of previous traversal for $j-1$ in the current traversal for j. Say that we store the values of indexes of the paths from the first column to row $j-2$ in G_P^j; then, the computation of *shift*(j) reduces to computing from which nodes in row $j-2$ we can reach row j. This is a constant-cost computation since the branching factor for each node is at most 3. Thus, the complexity of computing *shift*(j) reduces to $O(m^2)$.

5.3 Aggregates in the Star

There are two kinds of aggregates associated with the star. The first kind is the continuous aggregates that return a new value for each new input tuple as `ccount(X)` of Example 2.6. The second is the final aggregates that are computed at the end of the star sequence, such as `count(*A)` that in Example 2.5. The different syntax, whereby a starred variable is used for the second case, clearly denotes that the aggregate value for this second case is only available for the whole sequence, rather than for each individual value.

The standard OPS algorithm can be used to optimize the search for patterns specified via conditions on aggregates; this can be done by simply including new virtual attributes in the tuples being searched—one new attribute for each aggregate. Thus, for the query in Example 2.6, we add the distinguished columns `ccount` and `first` to `X`, to support the two aggregates `ccount(X)` and `first(X)`. At runtime, the values of attributes `ccount` and `first` are updated by their respective aggregate functions, for each new input in the star sequence. At compile time, however, the OPS algorithm treats them in the optimization process as any other attribute. Thus, the query of Example 2.6 is executed and optimized as follows:

```
SELECT  Y.SessNo
FROM Sessions
     CLUSTER BY  SessNO
     SEQUENCE BY  ClickTime
     AS (*X, Y)
```

```
WHERE A.PageType <>‘p’
 AND  X.ccount < 100
 AND  X.first + 20 Minute >
      X.ClickTime AND  Y.PageType<>‘p’
```

Final aggregates are also treated via the addition of new virtual attributes to the tuples being scanned; however the conditions involving these aggregates are tested in the pattern element following the star in which they were computed (if the star was the last element in the pattern a new dummy element is added to enable this test). Thus, the query in Example 2.5 is compiled as follows:

```
SELECT  SessNo, B.count
FROM Sessions
     CLUSTER BY  SessNO
     SEQUENCE BY  ClickTime
     AS (*A, B)
WHERE A.PageType <> ‘d’
  AND B.PageType = ‘d’
  AND B.count < 20
```

Thus, the original condition count(∗A) < 20 is implemented by the condition B.count < 20, which checks at point B if the value left in count by ∗A is in fact less than 20.

In summary, our general approach to compilation and execution uses a finite-state model, whereby there are as many states as there are variables in the search pattern. Therefore, the conditions in the WHERE clause are partitioned accordingly: the jth group in the partition contains all the clauses that actually use the jth variable, and to not use any later variable. For instance, in Example 2.5 there are two states each with its own conditions: the first is for variable A and the other for variable ∗B. In Example 3.2, there are five variables (but in our discussion we ignored the first one that plays no role in the implications).

6. EXPERIMENTAL RESULTS

In order to verify the applicability of our algorithm in real life application, we wrote an emulation in Matlab to measure the possible speedup when searching some popular patterns in stock market data. The speedups obtained range from the modest one obtained for the simple search pattern of Figure 5, 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 the so-called “double-bottom pattern”, that is, for the situation where the price of a stock has two consecutive local minima. 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 7).

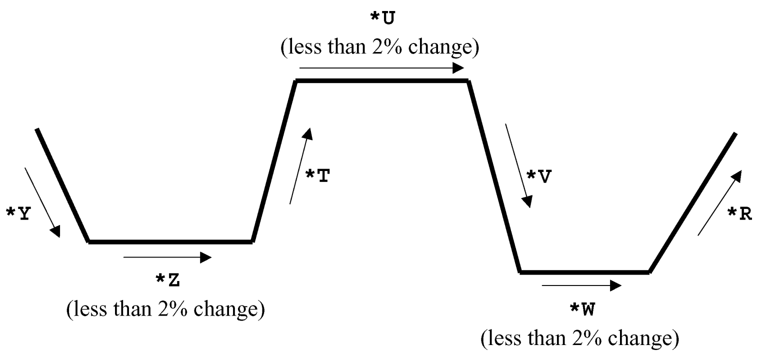

Fig. 7. The relaxed double bottom pattern.

Example 6.1. 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
```

Example 6.1 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 7). This query, optimized using the OPS algorithm, executes 93 times faster than the naive execution on the DJIA's data for the last 25 years. Figure 8 shows there are 12 matches found in the input. The graph in the bottom of Figure 8 shows one of these patterns that occurred around June 1990. We ran several queries with more complex search patterns, and measured speedups up to 800 times over naive search.

7. DISJUNCTIVE PATTERNS AND OTHER EXTENSIONS

The basic optimization approach of OPS is quite robust and can be extended to deal with different situations—in particular with disjunctive patterns, discussed next.

7.1 Calculating θ and ϕ for Disjunctive Predicates

One first situation is that of Example 7.1, where we have a conjunctive pattern where disjunctive conditions are applied to individual elements in the pattern.

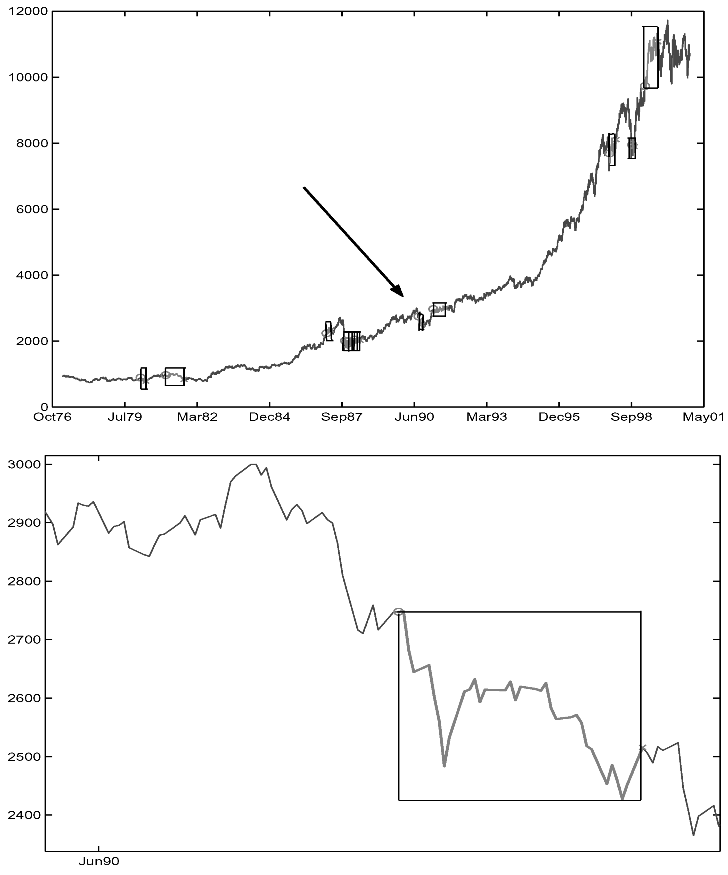

Fig. 8. Doublebottoms found in the DJIA data are shown by boxes. The bottom picture is zoomed for the area pointed by arrow in the top picture and shows one of the matches.

We will next discuss how to solve the problem of determining the ϕ and θ matrices for this situations. Assume we have the pattern $P = p_1, p_2, p_3, \ldots, p_m$, where, for some $1 \leq i \leq m$, the predicate p_i is disjunctive. To calculate the implication and satisfiability relations between predicate p_j and predicate p_k where $1 \leq k < j$, we must consider two (nonmutually exclusive) situations. The first situation is when $p_j = p_{ja} \vee p_{jb}$, and we want to compute θ_{jk}, that is, the logical value of $p_j \Rightarrow p_k$. If $p_{ja} \Rightarrow p_k$ and $p_{jb} \Rightarrow p_k$ are both true, $p_j \Rightarrow p_k$ is true and the value of θ_{jk} is 1. In a similar way, if $p_{ja} \Rightarrow \neg p_k$ and $p_{jb} \Rightarrow \neg p_k$, then θ_{jk} is 0. In the remaining cases, the information available is not enough and we need to set the value of θ_{jk} to U.

The second situation is when $p_k = p_{ka} \vee p_{kb}$ and we need to calculate the truth value of $p_k \Rightarrow p_j$. In this case, if either $p_j \Rightarrow p_{ka}$ or $p_j \Rightarrow p_{kb}$ have a truth value of 1, then $p_j \Rightarrow p_k$ is true and the value of $\theta_{jk} = 1$. Also, since $\neg p_k = \neg(p_{ka} \vee p_{kb}) = \neg p_{ka} \wedge \neg p_{kb}$, if $p_j \Rightarrow \neg p_{ka}$ and $p_j \Rightarrow \neg p_{kb}$ are both true, then we can conclude that $p_j \Rightarrow \neg p_k$ is true and θ_{jk} is 0.

Table I. Logic Matrix Elements for Disjunction of Pattern Elements

$p_j = p_{ja} \vee p_{jb}$	$\theta_{jb,k} = 1$	$\wedge$	$\theta_{jk} = 1$	$\theta_{jb,k} = 1$
	$\theta_{ja,k} = 0$	$\wedge$	$\theta_{jb,k} = 0$	$\theta_{jk} = 0$
	else			$\theta_{jk} = U$
	$\phi_{ja,k} = 1$	$\vee$	$\phi_{jb,k} = 1$	$\phi_{jk} = 1$
	$\phi_{ja,k} = 0$	$\vee$	$\phi_{jb,k} = 0$	$\phi_{jk} = 0$
	$\phi_{ja,k} = U$	$\wedge$	$\phi_{jb,k} = U$	$\phi_{jk} = U$
$p_k = p_{ka} \vee p_{kb}$	$\theta_{j,ka} = 1$	$\vee$	$\theta_{j,kb} = 1$	$\theta_{jk} = 1$
	$\theta_{j,ka} = 0$	$\wedge$	$\theta_{j,kb} = 0$	$\theta_{jk} = 0$
	else			$\theta_{jk} = U$
	$\phi_{j,ka} = 1$	$\vee$	$\phi_{j,kb} = 1$	$\phi_{jk} = 1$
	$\phi_{j,ka} = 0$	$\vee$	$\phi_{j,kb} = 0$	$\phi_{jk} = 0$
	else			$\phi_{jk} = U$

The same arguments can be used for calculating ϕ. For calculating ϕ_{jk} where $k < j$, we want to see if value of $\neg p_j \Rightarrow p_k$ is always true or not. Since $\neg p_j = \neg(p_{ja} \vee p_{jb}) = \neg p_{ja} \wedge \neg p_{jb}$, if $\neg p_{ja} \Rightarrow \neg p_k$ or $\neg p_{jb} \Rightarrow \neg p_k$, then we can conclude that ϕ_{jk} is 1.

Table I summarizes different possibilities for θ and ϕ. In this table, by $\theta_{ja,k}$ ($\theta_{jb,k}$) we mean value of θ_{jk} if we would replace p_j with p_{ja} (p_{jb}). The same notational conventions are used for ϕ. When we calculate θ_{jk}, and both p_j and p_k are disjunctions, we can first decompose the predicate and use the technique just described for calculating θ, and then combine the results. Obviously, we can use the same technique for calculating ϕ. Furthermore, the technique is easily generalized to the situation of disjunctions of more than two terms. As an example of disjunctive elements, let's consider the following SQL-TS query:

Example 7.1. Query with Disjunctions

```
SELECT X.NEXT.date, X.NEXT.price,
       S.previous.date, S.previous.price
FROM djia
   SEQUENCE BY date
   AS (X,*Y, Z, *T,U )
WHERE  X.price > 0.98 * X.previous.price
   AND Y.price < 0.98 * Y.previous.price
   AND (0.98*Z.previous.price < Z.price OR Z.price > 50)
   AND T.price > 1.02 * T.previous.price
   AND (0.98*U.previous.price < U.price OR U.price > 50)
```

This can be represented as a pattern sequence $P = p_1, p_2, p_3, p_4, p_5$, where:

$$p_1(t) = t.price > 0.98 * t.previous.price$$
$$p_2(t) = t.price < 0.98 * t.previous.price$$
$$p_3(t) = t.price > 0.98 * t.previous.price \vee t.price > 50$$
$$p_4(t) = t.price < 0.98 * t.previous.price$$
$$p_5(t) = t.price > 0.98 * t.previous.price \vee t.price > 50.$$

We now have $p_3 = p_{3a} \vee p_{3b}$ and $p_5 = p_{5a} \vee p_{5b}$ where:

$p_{3a}(t) = t.price > 0.98 * t.previous.price$
$p_{5a}(t) = t.price > 0.98 * t.previous.price$
$p_{3b}(t) = t.price > 50$
$p_{5b}(t) = t.price > 50.$

For instance, to calculate θ_{32}, notice that $\theta_{3a,2}$ (i.e., $p_{3a}(t) \Rightarrow p_2(t)$), $\theta_{3b,2}$ (i.e., $p_{3b}(t) \Rightarrow p_2(t)$) are not equal to 1. Moreover, $\theta_{3a,2} = 0$ (since $p_{3b}(t) \Rightarrow \neg p_2(t)$), but $\theta_{3b,2}$(i.e., $p_{3b}(t) \Rightarrow \neg p_2(t)$) is not 0. Thus, $\theta_{32} = U$. In conclusion, we obtain the following matrix for the query in Example 7.1.

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

7.2 Disjunctive Patterns

We next consider queries as that in Example 7.2 that searches for the disjunction of two patterns, where the input sequence can satisfy either one pattern or both. In effect, this query is equivalent to two independent queries, but we can execute them in one scan of the database.

A naive approach in processing the query of Example 7.2 would consist in making a first pass through data to satisfy the first pattern followed by a second pass to satisfy the second pattern. This approach will not be considered since it is likely to require each page in the secondary store to be retrieved twice.

Example 7.2. Find four consecutive rise in the stock price or four consecutive closing price between 55 and 57 for IBM

```
SELECT X.NEXT.date, X.NEXT.price,
       S.previous.date, S.previous.price
FROM quote
   SEQUENCE BY date
   AS (X, Y, Z, T)
WHERE X.name='IBM'
   AND (  (X.previous.price < X.price
           AND X.price < Y.price
           AND Y.price < Z.price
           AND Z.price < T.price )
        OR ( X.price > 55
            AND X.price < 57
            AND Y.price > 55
            AND Y.price < 57
            AND Z.price > 55
            AND Z.price < 57
            AND T.price > 55
            AND T.price < 57)
        )
```

We next consider two other approaches that do not suffer from this drawback. These are:

—Multiple Stream model, and
—Single Stream model.

7.2.1 *Multiple Data Streams.* In this model, cursors on the input and on the pattern, are kept are kept in a queue for each disjunctive pattern. When the pattern being tested fails, its values are computed and replace the old values in the queue. Then, the scheduler looks at the values in the queue and select a pattern for processing according to some optimization criteria. The queue can be prioritized based on different criteria for optimization. For instance, by selecting the pattern which has the least value of i, we can ensure that all patterns are served fairly and no pattern stays behind. This, in turn, minimizes the size of data that needs to be kept in temporary memory.

An advantage of this method is its simplicity and amenability to both serial and parallel processing. It can be implemented as a client-server model where the server provides the next values for i and j to each client process. Thus, this method is amenable to parallel execution based on multiple data streams.

7.2.2 *Single Data Stream.* This model assumes that all the patterns are tested in parallel against the current element in the input being scanned. The only patterns excluded are those already known to be false or true. Take the query in Example 7.2. That query is equivalent with two queries, one that finds occurrences with four consecutive closing prices between 55 and 57 and the other that finds four consecutive rise in the price. We calculate θ, ϕ, *shift* and *next* independently for each pattern, but at the run time we handle both queries simultaneously. The run-time algorithm can be revised as shown in Algorithm 7.3.

As the algorithm shows, we keep proceeding until one of the concurrent patterns fails. At this point we save the current value of i in i_O and reset i and j for the failed pattern and keep searching only for the fail pattern until i becomes greater than i_O (since we know the other pattern doesn't have to checked against input up to the point i_O). In this way, we only scan the pattern once and we need only one buffer to keep recent values of the input.

In Example 7.2, we have one pattern in the FROM clause, and disjunctive conditions in the FROM clause. The optimization and execution techniques discussed here, however, apply to more general situations where, for example, we have multiple SQL-TS queries with distinct FROM and WHERE—provided that the queries use the same CLUSTER BY and SEQUENCE BY clauses. In general, similar techniques might be applicable to the optimization of a set of continuous queries in a data streaming environment [Babcock et al. 2002]. The situation of disjunction in predicates discussed in the previous section can also be reduced to this, by simply normalizing them into disjunctive normal form. This approach is appealing as long as it does not lead to too many alternative patterns. The study of these extensions and improvements has been left for further research.

Algorithm 7.3. The OPS Algorithm for Concurrent Disjunctive Patterns

```
$j_1 = 1;$  $j_2 = 1;$  $i = 1;$
while $j_1 \leq m_1 \wedge j_2 \leq m_2 \wedge i \leq n$ do {
   while $p_{j_1}(t_i) \wedge q_{j_2}(t_i)$ do {
      $j_1 = j_1 + 1;$  $j_2 = j_2 + 1;$  $i = i + 1;$
   }
   $i_O = i;$
   if $\neg p_{j_1}(t_i)$ then {
      while $i \leq i_O$ do {
         while $j > 0 \wedge \neg p_{j_1}(t_i)$ do {
            $i = i - j_1 + next_1(j_1) + shift_1(j_1);$
            $j_1 = next_1(j_1);$
         }
         $j_1 = j_1 + 1;$  $i = i + 1;$
      }
   else {      $/* \neg p_{j_2}(t_i) */$
      while $i \leq i_O$ do {
         while $j > 0 \wedge \neg q_{j_2}(t_i)$ do {
            $i = i - j_2 + next_2(j_2) + shift_2(j_2);$
            $j_2 = next_2(j_2);$
         }
         $j_2 = j_2 + 1;$  $i = i + 1;$
      }
   }
}
if $i > n$ then failure
else success;
```

7.2.3 *More General Predicates.* 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) is given in Sadri [2001]. Said 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 [Sadri 2001].

8. CONCLUSIONS

We have described a novel approach for querying complex sequential patterns and optimizing these queries. By adding minimal syntactic extensions to SQL, we introduced SQL-TS, whose power in querying complex sequential patterns was then demonstrated with several examples. Furthermore, we proposed a method for optimizing queries that involve complex sequential patterns. Our method uses the logical interdependencies between the elements of the pattern to avoid repetitive scans over the input. In addition, we extended our optimization method to cover repetitive patterns (star patterns), arbitrary aggregates (including user-defined ones) and disjunctive patterns. The results of our experiments show substantial speedups for practical applications.

Many interesting research problems not discussed in this paper deserve further investigation. One is the problem of extending the applicability of the approach. For instance, in Sadri [2001] we focused on calculating ϕ and θ for more general classes of predicates; thus techniques were proposed to deal with containment predicates for intervals (including multidimensional ones), as needed to the search for patterns in a spatio-temporal database [Sadri 2001]. Another interesting problem is to extend SQL-TS to deal with noise, scaling, and similarity queries [Rafiei and Mendelzon 2000].

In terms of performance, for instance, many indexing techniques for sequences have been proposed recently [Ferragina et al. 2001; Wang and Perng 2003], and their use in conjunction with the optimization techniques here discussed deserves an in-depth investigation. Also, generalizations similar to those we have discussed for KMP should be investigated for other pattern search algorithms. Although there is evidence that KMP provides excellent performance on the average [Wright et al. 1998], other algorithms, such as those by Karp and Rabin [1987] and Boyer and Moore [1977], can offer performance advantages in particular situations.

The fast emerging area of data streams [Carney et al. 2002; Arasu et al. 2002; Chandrasekaran et al. 2003] provides many opportunities. The language extensions featured in SQL-TS provides capabilities for pattern matching and performing approximate (nonblocking) aggregates that are needed for streams. However, the execution and optimization strategies must be revisited in the light of the unique requirements of streams, where, for example, the use of memory must be optimized along with the cost of processing.

APPENDIX: The Syntax of SQL-TS

The SQL-TS queries consists of the basic select-from-where clauses of SQL, where the from clause is extended to support the sequence definition.

```
    ⟨sqlts-query⟩  ⟶  ⟨select-clause⟩
                      ⟨from-clause⟩
                      ⟨where-clause⟩
    ⟨from-clause⟩  ⟶  FROM⟨sequence⟩, ⟨sql-from-list⟩
       ⟨sequence⟩  ⟶  ⟨sql-table⟩|⟨sql-table-expression⟩
                      [CLUSTER BY⟨expression⟩{, ⟨expression⟩}]
                      [SEQUENCE BY⟨expression⟩{, ⟨expression⟩}
                                 [⟨ordering⟩] [⟨null-ordering⟩]]
                                 [AS⟨sid⟩|(⟨sid⟩{, ⟨sid⟩})]
            ⟨sid⟩  ⟶  ⟨id⟩|∗⟨id⟩
      ⟨ϵordering⟩  ⟶  ASC|DESC
  ⟨null-ordering⟩  ⟶  NULLS FIRST|NULLS LAST
```

Here ‘sql-from-list’ denotes the list of table names, or table expressions supported in standard SQL. Thus, an SQL-TS can use any number of tables, but the sequence pattern can only be applied to the first table in the from clause (this could actually be a derived table constructed via a table expression). Note that when the SEQUENCE BY and CLUSTER BY clauses are empty, the AS can work for making aliases.

ACKNOWLEDGMENTS

We would like to thank the reviewers for the improvements they have suggested on the original manuscript. Thanks are also due to Xin Zhou and Leila Kaghazian for very helpful comments, and to Hetal Thakkar for implementing the SQL-TS prototype.

REFERENCES

AGRAWAL, R. AND SRIKANT, R. 1995. Mining sequential patterns. In *Proceedings of the International Conference on Data Engineering*.

ALUR, N., HAAS, P., MOMIROSKA, D., READ, P., SUMMERS, N., TOTANES, V., AND ZUZARTE, C. 2002. Db2 UDBS high-function business intelligence in e-business. IBM redbooks, IBM, `http://www.redbooks.ibm.com/redbooks/pdfs/sg246546.pdf`.

ARASU, A., BABU, S., AND WIDOM, J. 2002. An abstract semantics and concrete language for continuous queries over streams and relations. Tech. rep., Stanford Univ., Stanford, Calif.

BABCOCK, B., BABU, S., DATAR, M., MOTAWANI, R., AND WIDOM, J. 2002. Models and issues in data stream systems. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, New York, 1–16.

BERRY, M. AND LINOFF, M. 1996. *Data Mining Techniques for Marketing, Sales, and Customer Support*. Wiley, New York.

BOAG, S., CHAMBERLIN, D., FERNANDEZ, M. F., FLORESCU, D., ROBIE, J., AND SIMEON, J. (EDS.). 2003. Xquery 1.0: An XML query language–working draft 22 august 2003. Working Draft 22 August 2003, W3C, `http://www.w3.org/tr/xquery/`.

BOYER, R. S. AND MOORE, J. S. 1977. A fast string searching algorithm. *Commun. ACM 20*, 10, 762–772.

CARNEY, D., CETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., SEIDMAN, G., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. 2002. Monitoring streams—A new class of data management applications. In *VLDB* (Hong Kong, China).

CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S. R., RAMAN, V., REISS, F., AND SHAH, M. A. 2003. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR 2003: First Biennial Conference on Innovative Data Systems Research* (Asilomar, Calif., Jan. 5–8).

EDWARDS, R. AND MAGEE, J. 1997. *Technical Analysis of Stock Trends*. AMACOM.

FALOUTSOS, C., RANGANATHAN, M., AND MANOLOPOULOS, Y. 1994. Fast subsequence matching in time-series databases. In *Proceedings of the International Conference on Management of Data*. 419–429.

FERRAGINA, P., KOUDAS, N., MUTHUKRISHNAN, S., AND SRIVASTAVA, D. 2001. Two-dimensional substring indexing. In *Proceedings of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Santa Barbara, Calif., May 21–23). ACM, New York.

GATZIU, S. AND DITTRICH, K. R. 1993. Events in an object-oriented database system. In *Proceedings of the 1st International Conference on Rules in Database Systems*.

GEHANI, N. H., JAGADISH, H. V., AND SHMUELI, O. 1992. Composite event specification in active databases: Model and implementation. In *Proceedings of the 18th International Conference on Very Large Data Bases*.

GUO, S., SUN, W., AND WEISS, M. 1996a. On satisfiability, equivalence, and implication problems involving conjunctive queries in database systems. *IEEE Trans. Knowl. Data Eng. 8*, 4, 604–616.

GUO, S., SUN, W., AND WEISS, M. A. 1996b. Solving satisfiability and implication problems in database systems. *ACM Trans. Datab. Syst. 21*, 2, 270–293.

HELLERSTEIN, J. M., HASS, P. J., AND WANG, H. J. 1997. Online aggregation. In *Proceedings of the International Conference on Management of Data*. ACM, New York, 171–182.

INFORMIX SOFTWARE, INC. 1998. Managing time-series data in financial applications. White Paper.

KARP, R. AND RABIN, M. O. 1987. Efficient randomized pattern matching algorithms. *IBM J. Res. Devel. 31*, 2 (Mar.), 249–260.

KLUG, A. 1988. On conjunctive queries containing inequalities. *J. ACM 35*, 1, 146–160.

KNUTH, D. E., MORRIS, J. H., AND PRATT, V. R. 1997. Fast pattern matching in strings. *SIAM J. Comput. 6*, 2 (June), 323–350.

MESROBIAN, E., MUNTZ, R., SANTOS, J., SHEK, E., MECHOSO, C., FARRARA, J., AND STOLORZ, P. 1994. Extracting spatio-temporal patterns from geoscience datasets. In *Proceedings of the IEEE Workshop on Visualization and Machine Vision*.

MOTAKIS, I. AND ZANIOLO, C. 1997. Temporal aggregation in active databases. In *Int. Conf. on the Managment of Data*.

PERNG, C.-S. AND PARKER, D. S. 1999. SQL/LPP: A time series extension of SQL based on limited patience patterns. In *Database and Expert Systems Applications, 10th International Conference (DEXA '99)*. Lecture Notes in Computer Science, vol. 1677. Springer-Verlag, New York.

RAFIEI, D. AND MENDELZON, A. 2000. Querying time series data based on similarity. *TKDE 12*, 5, 675–693.

RAMAKRISHNAN, R., DONJERKOVIC, D., RANGANATHAN, A., BEYER, K., AND KRISHNAPRASAD, M. 1998. SRQL: Sorted relational query language. In *Proceedings of the 10th Annual International Conference on Scientific and Statistical Database Management* (Capri, Italy, July 1–3). IEEE Computer Society Press, Los Alamitos, Calif., 84–95.

ROSENKRANTZ, D. AND HUNT III, H. B. 1970. Processing conjunctive predicates and queries. In *Proceedings of the 6th International Conference on Very Large Databases*. 64–72.

SADRI, R. 2001. Optimization of sequence queries in database systems. Ph.D. thesis, University of California, Los Angeles.

SESHADRI, P. 1998. Predator: A resource for database research. *SIGMOD Record 27*, 1, 16–20.

SESHADRI, P., LINVY, M., AND RAMAKRISHNAN, R. 1994. Sequence query processing. In *Proceedings of ACM SIGMOD Conference on Management of Data*. ACM, New York, 430–441.

SESHADRI, P., LIVNY, M., AND RAMAKRISHNAN, R. 1995. SEQ: A model for sequence databases. In *ICDE*. 232–239.

SUN, W. AND YU, C. 1994. Semantic query optimization for tree and chain queries. *IEEE Trans. Knowl. Data Eng. 6*, 5.

SUN, X., KAMELL, N. N., AND NI, L. M. 1989. Processing implication on queries. *IEEE Trans. Softw. Eng. 5*, 10 (Oct.), 168–175.

ULLMAN, J. D. 1989. *Principles of Database and Knowledge-Base Systems*. Vol. 2. Computer Science Press.

WANG, H. AND PERNG, C. 2003. The s2-tree: An index structure for subsequence matching of spatial objects. In *Proceedings of the 5th Pacific-Asic Conference on Knowledge Discovery and Data Mining (PAKDD)* (Hong Kong, China).

WANG, H. AND ZANIOLO, C. 2000. Using SQL to build new aggregates and extenders for object-relational systems. In *Proceedings of 26th International Conference on Very Large Data Bases*.

WRIGHT, C. A., CUMBERLAND, L., AND FENG, Y. 1998. A performance comparison between five string pattern matching algorithms. Technical Report. http://ocean.st.usm.edu/~cawright/pattern_matching.html.

ZEMKE, F., KULKARNI, K., WITKOWSKI, A., AND LYLE, B. 1999. Proposal for OLAP functions. Tech. Rep. ANSI NCITS H2-99-155, ISO/IEC JTC1/SC32 WG3:YGJ-nnn, http://ganga.iiml.ac.in/~bhasker/dmds/99-154.pdf.

Received December 2002; revised June 2003; accepted October 2003