Journal of Intelligent Information Systems, 15, 29–49, 2000

Temporal Coupling Verification in Time Series Databases

CHANG-SHING PERNG perng@cs.ucla.edu
D. STOTT PARKER stott@cs.ucla.edu
Department of Computer Science, University of California, Los Angeles, USA

Abstract. Time series are often generated by continuous sampling or measurement of natural or social phenomena. In many cases, events cannot be represented by individual records, but instead must be represented by time series segments (temporal intervals). A consequence of this segment-based approach is that the analysis of events is reduced to analysis of occurrences of time series patterns that match segments representing the events.

A major obstacle on the path toward event analysis is the lack of query languages for expressing interesting time series patterns. We have introduced SQL/LPP (Perng and Parker, 1999). Which provides fairly strong expressive power for time series pattern queries, and are now able to attack the problem of specifying queries that analyze temporal coupling, i.e., temporal relationships obeyed by occurrences of two or more patterns.

In this paper, we propose SQL/LPP+, a temporal coupling verification language for time series databases. Based on the pattern definition language of SQL/LPP (Perng and Parker, 1999), SQL/LPP+ enables users to specify a query that looks for occurrences of a cascade of multiple patterns using one or more of Allen's temporal relationships (Allen, 1983) and obtain desired aggregates or meta-aggregates of the composition. Issues of pattern composition control are also discussed.

Keywords: temporal coupling verification, time series databases, meta-aggregation, time series patterns, causality

1. Introduction

1.1. Motivation

Discovering temporal coupling among events is a fundamental method for detecting causality among events. The study of causality is considered to be one of the most important tasks for most natural and social scientists. Although there are many different approaches to identify probable cause and effect among events, perhaps the simplest and the most intuitive is to observe historical data and check the likelihood that a set of events occurs in a particular temporal order. When the occurrence of events is not predictable, continuous observation or measurement of the subject phenomenon is a common way to track events. For example, Richter readings and stock index values are continuously recorded in order to catch important seismic and economic events respectively. The result of these observations and measurements is time series data.[1] Hence time series data is often the only scientific ground on which to build theories. Events, in this representation, are *time series patterns*

which possess some special features. For example, El Niño and La Niña are recognized by rising and falling sea water temperatures of in the South Pacific.

We define *temporal coupling* as the likelihood that a given set of events occurs in a particular time period. To extract knowledge from data, domain experts first propose hypotheses about the temporal coupling of events, then develop a specialized program to verify their hypotheses. *Temporal coupling verification* (*TCV*) is the task of confirming hypothesized temporal coupling among events.

For example, we would like to know how likely it is that the DJIA will decline after a period of increasing interest rates, and how likely the total seasonal rainfall in Los Angeles will increase after a 3-month period of increasing sea water temperature. While establishing the real cause and effect relations among these events requires profound domain knowledge, TCV does not, and it can be automated. We believe temporal coupling verification is a necessary part of theory construction, and researchers can benefit from utilizing temporal coupling verification languages.

1.2. Motivating examples

Example 1. This example is a typical case in technical analysis of stocks (Edwards and Magee, 1997). A *Double Bottom* formation is commonly seen as a pattern that signals the onset of an uptrend. Short-term stock traders might be interested to know what their average profit, maximal loss and total number of trades would have been, if they had simply bought a stock when they had seen the pattern form and then held it for a fixed number of trading days.

Example 2. An investor in the stock market may be interested to know, based on financial history, how likely it is that a downtrend of the DJIA will last for more than 6 months during periods that the 30-year T-bond yield was below 5.0%.

Example 3. Economists might want to know from history how many bear markets (when the DJIA falls 20% from its previous high) were preceded by a period of high interest rates.

Example 4. Seismologists might be interested to know whether major earthquakes were always preceded by a number of smaller earthquakes.

It is easy to see that these examples have common features. In each case, the events of interest are not individual records. Instead, they are represented by a group of records in continuous time intervals. The temporal coupling among the events can be verified by counting the occurrences of the corresponding time series patterns. With currently available technology, answering each question requires tedious procedural program coding. Considering the fact that these hypothesis tests are usually run only once, or need frequent modification, the cost of this kind of knowledge extraction is fairly high. If there were a formal language that could express all these patterns, then domain experts would no longer need to spend valuable time in coding/debugging and could concentrate more on perfecting their theories. Computer scientists could also focus on improving the language and its execution efficiency instead of helping domain experts in a case-by-case manner. The need for a declarative language that permits fast formulation and execution of queries is obvious.

1.3. The problems

In the aspect of data granularity, knowledge discovery in databases (KDD) and data mining (Fayyad et al., 1996) in time series differ from those in set-oriented data. As stated by Shatkay (Shatkay and Zdonik, 1996), in time series

> individual values are usually not important but the relationships between them are.

It is thus often continuous time series segments, instead of individual records, that represent events. Any attempt to design a time series data mining language must provide a way to define patterns that represent events.

Time series patterns can be defined illustratively or descriptively. Illustrative patterns are defined by drawing pictures or example segments. Descriptive patterns are defined by some formal language. To date, most current research has focused on illustrative pattern search. Both approaches have their own advantages and disadvantages. However, since illustrative pattern definition usually does not come with flexible similarity measurement, the specified patterns might often fail to express the real intent of users. We argue that the descriptive pattern approach can better express users' intended patterns because they allow users to specify the *features* and similarity measurements they want. For example, illustrative patterns cannot express concepts like volatility, relative strength and crossing moving averages which are commonly used in financial data analysis.[2] We only discuss descriptive patterns in this paper.

From the viewpoint of KDD and data mining, Shatkay's statement can be further improved to:

> Individual time series segments are usually not important, but the *coupling relationships* between them are.

It is very hard for the human mind to learn anything from a large number of events. Another level of abstraction is required. For each of the motivating examples, the answer should consist of only one or a few numerical values instead of a listing of all related events. *The purpose of TCV query languages is to provide that extra level of abstraction.*

Three issues surface immediately when one attempts to discover temporal coupling among events:

1. An adequate database model is required to store and manipulate time series data. Time series data is inherently linearly ordered. The set-oriented relational model does not reasonably capture this.
2. An expressive and efficiently executable pattern definition language is required to define time series patterns.
3. A data mining language is required to formulate the temporal coupling verification task.

In the next subsection we will discuss the current status of work on these issues.

1.4. Related work

As indicated in (Schmidt et al., 1995), until recently time series databases had been long a neglected issue in database research. Since then, research has advanced in many directions. Some data representation models have been proposed (Segev and Soshani, 1993; Seshadri et al., 1994, 1995; Informix, 1994) which address the sequentiality of time series data and provide storage models. On the issue of pattern query, research following the illustrative approach has continuously extended the definition of similarity from exact match (Faloutsos et al., 1994) to allow some degree of flexibility (Berndt and Clifford, 1996; Lin and Risch, 1998). Following the descriptive approach, TREPL (Motakis and Zaniolo, 1997), SDL (Agrawal et al., 1995) and Sequence Datalog (Mecca and Bonner, 1995) are a few examples of time series-related query languages developed before the introduction of SQL/LPP (Perng and Parker, 1999). The well-known temporal query language TSQL2 (Snodgrass, 1995) does not offer any way to define general patterns. To our best knowledge, there is no query language specifically designed for knowledge discovery and data mining in time series databases.

1.5. The basic idea

Recognizing the importance of TCV, in this paper we attempt to generalize the problem and design a formal language that is expressive enough to address TCV problems in an unambiguous manner. The language we propose, SQL/LPP+, serves as a problem definition language for TCV problems as VHDL serves for hardware description and XML for documentation. Although the LPP model (Perng and Parker, 1999) provides a fairly efficient execution model, we do not exclude the possibility that future research can perform with greater efficiency.

We believe that a temporal coupling verification language should provide users a way to specify:

1. The (one or more) patterns of interest.
2. The temporal coupling relationship(s) among them.
3. The aggregates (statistics) of interest, which might be simple aggregates or meta-aggregates (Section 3.2).
4. The control of pattern occurrence counting, that is, whether an occurrence of a (syntactically) preceding pattern should couple with only one or multiple occurrences of the following pattern.

Our idea is to cascade pattern queries. Each pattern, with associated aggregates, is specified in a syntactic order and is connected by a combination of temporal relationships. The processing of the resulting query follows the syntactic order to find occurrences of each pattern and update aggregates associated with them. The final value of the aggregates is the output of the verification.

1.6. Outline of the paper

The language we propose, SQL/LPP+, is based on SQL/LPP's ability to define patterns. In Section 2, we discuss some basic issues in designing a time series pattern query language, e.g. segment ordering and redundancy reduction. Then we introduce the syntax of SQL/LPP+. By some examples, we show how to define SQL/LPP+ patterns and how to use the patterns to generate new time series. In Section 3, we present the language SQL/LPP+ by some examples. In Section 4, we conclude with a summary of our current status.

2. Defining SQL/LPP+ patterns

2.1. A formalization of time series database systems

SQL/LPP is a time series extension of SQL based on the *Limited Patience Patterns* (*LPP*) model. SQL/LPP+ is an extension of SQL/LPP. Since time series constitute a fairly new data type, we develop the concept of time series databases in a formalized way. First, we define time series elements, index functions and time series.

Definition 1. A *time series element* is an object of type $\mathcal{T} = int \times \tau_1 \times \cdots \times \tau_n$ where $n > 0$ and $\tau_1, \ldots, \tau_n$ are basic types. The first field of integer type is the *index* of the time series element. The columns of a time series element have names $c_1, \ldots, c_n$. The type of time series elements is written as $\{c_1 : \tau_1, \ldots, c_n : \tau_n\}$.

Definition 2. The function *index*: $\mathcal{T} \to int$ is defined as the projection function which projects the index from a time series element.

In practice, the index field of a time series element can be decided by the location of the element. So there is no need to actually record it. Also, the use of the index is only for the purpose of defining the language semantics so we often omit the field in the rest of the paper.

Definition 3. A *time series* is a function $ts : int \to \mathcal{T}$ where $\mathcal{T}$ is a time series element type and $index(ts(i)) = i$, and there exists an integer k, for every $i \leq k$, $ts(i)$ is defined and for every $i > k$, $ts(i)$ is undefined. The type of ts is denoted by $series(\mathcal{T})$. $ts(i)$ is usually denoted by $ts[i]$.

A time series element always belongs to a time series, so we can always omit the index field and make it implicit. This will not affect the index function because we can use notation like $index_{ts}(e)$ to represent the index of element e in time series ts.

The following definitions connect time series data with the relational model.

Definition 4. The *surrogate* of a time series is a tuple $\gamma_1 \times \cdots \times \gamma_m$ where $m > 0$ and $\gamma_1, \ldots, \gamma_m$ are basic types.

Definition 5. A *time series database* is a set of time series with distinct surrogates.

In this modeling of time series databases, one time series is placed in a field in a relational table. Surrogates are the keys of tables. As previously stated, time series segments are the subjects of queries.

Definition 6. A function s is a *time series segment* of length ℓ in a time series ts if there exists a integer $k > 0$ such that for all $1 \leq i \leq \ell$, $s[i] = ts[k + i - 1]$. The segment s is also denoted as $ts[k, k + \ell - 1]$. The type of segments is $seg(\mathcal{T}) = series(\mathcal{T}) \times int \times int$ where the two integers are the indices of the starting and ending elements.

The result of a pattern query is a group of segments, each of which can be seen as a *view*. Since there can be many such segments, it is important that they can be ordered.

Definition 7. The *segment ordering* $\ll$ is the lexicographic order of segments based on their starting and ending indices. That is,

$$\ll: seg(\mathcal{T}) \times seg(\mathcal{T}) \rightarrow \{0, 1\} \text{ such that}$$
$$ts[x_1, y_1] \ll ts[x_2, y_2] \Longleftrightarrow (x_1 < x_2) \vee (x_1 = x_2 \wedge y_1 < y_2)$$

Definition 8. The *maximal patience* is an integer L which represents a measurement of the resources available (for example, the number of time series elements that can fit in a computer's physical memory).

Definition 9. *Valid segments* are those segments with length no longer than the maximal patience.

Proposition 1. *Given a time series s and maximal patience L, the segment ordering on valid segments is a linear ordering.*

Now we formalize time series queries. By abusing language, we mix the notation of logic with that of boolean functions.

Definition 10. A *query sentence* $\mathcal{P}$ is a function of time series segments such that $\mathcal{P} : seg(\mathcal{T}) \rightarrow \{0, 1\}$.

Definition 11. A segment s is an answer to a query sentence $\mathcal{P}$ if $\mathcal{P}(s) = 1$. This is also denoted by $\mathcal{P} \models s$.

Even though segments are the subjects of time series queries, it is impractical to use whole segments in data manipulation or output because a segment may contain thousands or millions of time series elements. A few attributes are usually sufficient to represent a segment.

Definition 12. An attribute tuple *attr* is a function $attr : seg(\mathcal{T}) \rightarrow \mathcal{T}'$ where $\mathcal{T}$ and $\mathcal{T}'$ are element types.

Now we can proceed to define time series query processing systems.

Definition 13. A *time series query processing system* is a function $p : seg(\mathcal{T}) \times (seg(\mathcal{T}) \rightarrow \mathcal{T}') \times (seg(\mathcal{T}) \rightarrow \{0, 1\}) \rightarrow series(\mathcal{T}')$, such that given a time series *ts*, an attribute tuple *attr* and a query sentence $\mathcal{P}$, for every segment s of *ts*, $attr(s) \in p(ts, \mathcal{P})$ if $\mathcal{P} \models s$. Also, $index(attr(s_1)) < index(attr(s_2))$ if and only if $s_1 \ll s_2$.

Query processing works only on a single time series. In order to query multiple time series, we merge them into a single time series.

Definition 14. Assume $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$ are types of time series, the tuple concatenation operator $\uplus : \mathcal{T}_1 \times \mathcal{T}_2 \rightarrow \mathcal{T}_3$ is defined to contain the element $e_3 = e_1 \uplus e_2 = (c_{11} \ldots c_{2m_2})$ iff $e_1 = (c_{11}, \ldots, c_{1m_1})$ is of type $\mathcal{T}_1$, $e_2 = (c_{21}, \ldots, c_{2m_2})$ is of type $\mathcal{T}_2$ and $e_3 = (c_{11}, \ldots, c_{1m_1}, c_{21}, \ldots, c_{2m_2})$.

Definition 15. A time series *ts* is the *merge* of time series $ts_1, \ldots, ts_n$ by *synchronizing* on fields $c_1, \ldots, c_m$ if *ts* is the time series of maximal length such that every element e of *ts* is $e_1 \uplus \cdots \uplus e_n$, where $e_1.c_1 = \cdots = e_n.c_1, \ldots, e_1.c_m = \cdots = e_n.c_m$, $e_i \in ts_i$ for $1 \leq i \leq n$, and for any two elements e' and e'' of ts, $e' = e'_1 \uplus \cdots \uplus e'_n$, and $e'' = e''_1 \uplus \cdots \uplus e''_n$, $index(e') \leq index(e'')$ if and only if $index(e'_1) \leq index(e''_1), \ldots, index(e'_n) \leq index(e''_n)$.

2.2. SQL/LPP+ data definition language

The data definition language should be straightforward as demonstrated by the following example.

Example 5. Consider a daily stock database. The stock symbol serves as the surrogate. Each time series element contains 3 columns: date, closing price and trade volume. The table is declared as:

```
CREATE ROW TYPE quote(
  date datetime,
  price real,
  volume integer );

CREATE TABLE daily_stocks(
  symbol lvarchar,
  quotes TimeSeries(quote) );
```

The type `TimeSeries(quote)` represents a complex object which contains a sequence of records of `quote` type. An important feature about the data model is that even though we put a 'date' field in the row type, the model does not mandate any temporal data field. The temporal order is determined by an indexing function that represents the incoming order of real time data or the logical storage order.

All examples in the rest of the paper will be based on this schema.

Table 1. SQL/LPP+ functions.

Function	Definition
first	$first(s,k)=e$ iff $index_s(e)=k$.
last	$last(s,k)=e$ iff $1 \le \ell-k+1 \le \ell$ and $index_s(e)=\ell-k+1$.
count	$count(s)=\ell$.
sum	$sum(s,c_i)=r$ iff $r=\sum_{e\in s} e.c_i$.
avg	$avg(s,c_i)=t$ iff $t=\sum_{e\in s}\frac{e.c_i}{\ell}$.
max	$max(s,c_i)=u$ iff there exists an element e in s such that $e.c_i=u$ and for all e in s, $e.c_i \le u$.
min	$min(s,c_i)=v$ iff there exists an element e in s such that $e.c_i=v$ and for all e in s, $v \le e.c_i$.
is_first	$is\_first(e,k)=T$ iff $1 \le k \le \ell$ and $index_s(e)=k$.
is_last	$is\_last(e,k)=T$ iff $index_s(e)=\ell-k+1$.
next	$next(e_1,k)=e_2$ iff $index_s(e_1)=m$, $m+k \le \ell$ and $index_s(e_2)=m+k$.
prev	$prev(e_1,k)=e_2$ iff $index_s(e_1)=m$, $m-k \ge 1$ and $index_s(e_2)=m-k$.

Notation: e: a time series element, s: a time series segment, k: an integer, ℓ: the length of s, c_i: a column.

2.3. Defining SQL/LPP+ patterns

2.3.1. SQL/LPP+ functions. SQL/LPP+ functions are the core elements of SQL/LPP+. Some commonly used functions are listed in Table 1.

The SQL/LPP+ function set is extensible. Users can add more functions to the language if they feel it is necessary.

2.3.2. SQL/LPP+ sentences. An *atomic pattern formula* is a pattern expression with resulting type boolean. For example, the following pattern expressions are legal atomic pattern formulae:

- $last(s,3).price > 100$,
- $next(first(s,3),3).price - prev(last(s,2)).price > last(s).price$,
- $last(s).price > e.price$.

Pattern formulae are defined inductively:

1. An atomic pattern formula is a pattern formula.
2. If $\mathcal{P}$, $\mathcal{Q}$ are pattern formulae, then (*NOT* $\mathcal{P}$), ($\mathcal{P}$ *AND* $\mathcal{Q}$), ($\mathcal{P}$ *OR* $\mathcal{Q}$), [*ALL e IN s*]($\mathcal{P}$) and [*SOME e IN s*]($\mathcal{P}$) are all pattern formulae.

A pattern formula $\mathcal{P}$ is a *pattern sentence* if every element variable occurrence in $\mathcal{P}$ is quantified.

The interpretation of pattern sentences follows classical first order logic. That a segment $ts[x, y]$ satisfies a pattern sentence $\mathcal{P}$ is denoted by $ts[x, y] \models P$. Pattern interpretations must be *closed*, that is, if an atomic formula refers to any element not inside the segment, the atomic formula will not satisfy $\mathcal{P}$. For example, regardless of the content of the time series in question, we always have $ts[x, y] \not\models next(last(s)).price > 0$ and $ts[x, y] \models (NOT\ prev(first(s)).price > 0)$.

2.3.3. SQL/LPP+ search directives. An often encountered problem in pattern searching is that the search engine returns too many segments that either overlap, properly contain, or are adjacent to another. Consider an *uptrend* pattern in a daily stock price database. An uptrend is a continuous period that satisfies the following two conditions:

1. The closing price of each day, except the first day, is higher than the one of the previous day.
2. The length of the period is at least 5 days.

Assume there is a 10-day period in which a stock goes up every day. Then there are 21 segments that satisfy the conditions—one of length 10, two of length 9, ..., and six of length 5. In most applications, the number of answers is far more than desired. A pattern query language should provide something like user directives to limit the search for matched segments.

Based on the segment ordering shown in figure 1, SQL/LPP+ provides two classes of search directives. *Selection search directives* indicate which segments should be reported before searching the segments in the next row. *Restart search directives* specify where the search process should proceed after a matched segment is reported.

Assume the description of a pattern is a pattern sentence $\mathcal{P}$. The following are SQL/LPP+ selection search directives:

1. **MAXIMAL**: Given a starting point x, report only $ts[x, y]$ such that $ts[x, y] \models \mathcal{P}$ and for all $z > y$, $ts[x, z] \not\models \mathcal{P}$. This is a maximal length matched segment. Properly contained segments will not be reported. The length is bounded either by the specification of the pattern or the limit of system resources.

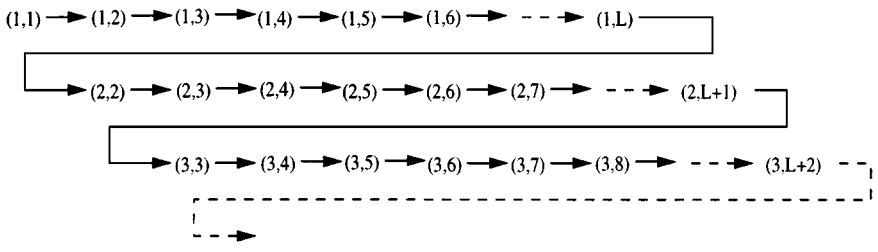

Figure 1. The segment space ordered by lexicographical ordering.

Date	1	2	3	4	5	6	7	8	9	10
Close	100	99	98	97	101	102	101	98	102	99
Segment	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)	(1,9)	(1,10)
Truth	F	F	F	F	**T**	T	**T**	F	**T**	F

Figure 2. The truth value shown reflects evaluation against the pattern sentence $first(s).price < last(s).price$. The segments selected with MINIMAL directive is $ts[1, 5]$, with FIRST MAXIMAL is $ts[1, 7]$ and with MAXIMAL is $ts[1, 9]$.

2. **FIRST MAXIMAL**: Given a starting point, report only the shortest segment $ts[x, y]$ such that $ts[x, y] \models \mathcal{P}$ but $ts[x, y+1] \not\models \mathcal{P}$. No proper sub-segment will be reported.
3. **MINIMAL**: Given a starting point, report the shortest matched segment.

These appear to cover the alternatives most useful in practice. Because they conflict mutually, only one of the above directives can be specified in a pattern.

Example 6. Consider a pattern **profitable_period** described by pattern sentence $first(s).price < last(s).price$. As shown in figure 2 different selection search directives select different segments.

Assume that $s = ts[x, y]$ is the last reported answer and n is a natural number. The following restart search directives indicate the restarting point. An illustrative explanation of restart search directives is in figure 3.

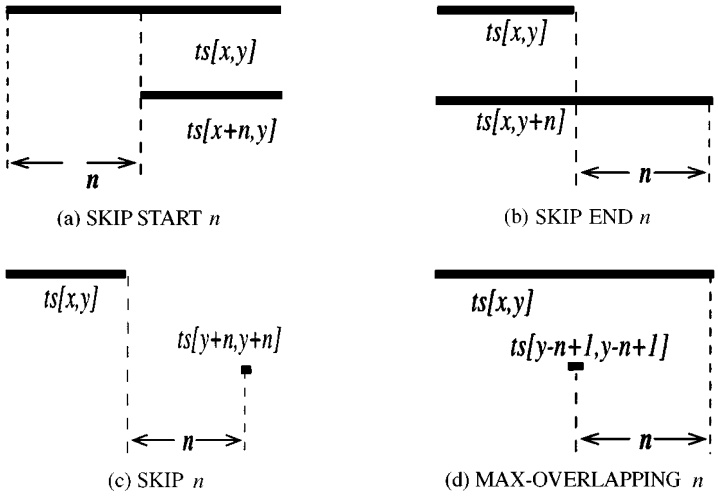

Figure 3. The relative position between the most recent reported segment and the search restarting point under different restart search directives.

1. **SKIP START** n: $ts[x+n, max(x+n, y)]$.
2. **SKIP END** n: $ts[x, y+n]$.
3. **SKIP** n: $ts[y+n, y+n]$.
4. **MAX-OVERLAPPING** n: $ts[y-n+1, y-n+1]$ if $y-n > x$, $succ(s)$ otherwise.
5. **ALL**: $succ(s)$.

Due to frequent use, we use **NON-OVERLAPPING** for `SKIP 1`. More than one of the restart search directives can be applied to a pattern. Suppose two directives specify $ts[x_1, y_1]$ and $ts[x_2, y_2]$ as the next starting point, then the search engine will restart at $ts[max(x_1, x_2), max(y_1, y_2)]$. For example, if the last reported answer is $ts[x, y]$ and the pattern is specified with `SKIP START 3` and `SKIP END 5`, the search will go on at $ts[x+3, y+5]$.

Back to the example of monotonic uptrend described previously. If the pattern is specified with `FIRST MAXIMAL`, only segment $ts[1, 10]$ will be reported. If it is specified with `MINIMAL` and `NON-OVERLAPPING`, two segments $ts[1, 5]$ and $ts[6, 10]$ will be reported.

2.3.4. Defining patterns. SQL/LPP+ patterns, like procedures, functions and triggers, are first-class objects in time series databases. A simple SQL/LPP+ pattern describes the properties of a single segment. In contrast, a composite SQL/LPP+ pattern is formed by many defined patterns.

2.3.5. Simple SQL/LPP+ pattern declaration. The main body of a simple SQL/LPP+ pattern is a segment of a certain element type. A number of public attributes can be defined by the ATTRIBUTE$\cdots$IS clause. Pattern sentences and search directives are placed in WHERE and WHICH_IS clauses respectively. The following examples demonstrate basic pattern declaration.

Example 7. The pattern *uptrend* described previously can be expressed as:

```
CREATE PATTERN uptrend AS
  SEGMENT s OF quote
    WHICH_IS FIRST MAXIMAL, NON-OVERLAPPING
  ATTRIBUTE date IS last(s).date
  ATTRIBUTE low  IS first(s).price
  ATTRIBUTE high IS last(s).price
  WHERE [all e in s](is_first(e)
          OR e.price >= prev(e).price)
        AND count(s) >= 5
```

This pattern has three public accessible attributes, *date*, *low* and *high*. The attributes are the only part other statements can access. The rest of the statement should be clear without explanation. The expressions $first(s).price$ and $last(s).price$ have lower overhead than $min(s, close)$ and $max(s, close)$, as it will soon become clear in following sections, though they are all constant-time computable where the buffer size is fixed.

Patterns can also be parameterized. The following example shows a parameterized pattern.

Example 8. The following pattern takes a period length as a parameter and finds segments of that length with date and moving average as attributes.

```
CREATE PATTERN moving_avg_seg(days int) AS
  SEGMENT s OF quote
  ATTRIBUTE date IS last(s).date
  ATTRIBUTE ma IS avg(s,close)
  WHERE count(s)=days
```

2.3.6. Composite pattern definition. Simple patterns do not permit the expression of partial aggregation such as *the average closing price of the first half period*. This shortcoming can be mended by composite patterns which are formed from many patterns. In the current design, only sequential composition is provided.

A composite pattern is declared as a concatenation of many patterns which are non-overlapping or overlapping on edges (having only one element in common). This is the only way to construct composite patterns in current design of SQL/LPP+ because we found alternative constructs are rarely needed and introduce high complexity into pattern query processing. The search directives of sub-patterns can be overridden by specifying new search directives. The syntax of non-overlapping composition is:

```
{pattern_1 alias_1 WHICH_IS search-directives,
 pattern_2 alias_2 WHICH_IS search-directives,
 ...
 pattern_n alias_n WHICH_IS search-directives}
```

Edge-overlapping composition is same as non-overlapping except that the separator ',' is replaced by ';'. In both cases the WHICH_IS phrases are optional. The following example demonstrates the use of composite patterns.

Example 9. Assume pattern *downtrend* is defined symmetrically to pattern *uptrend* in Example 7. The pattern *double_bottom* consists 4 trends as shown in figure 4. The pattern has following properties:

1. The starting point is 20% higher than the local maximum.
2. The difference of the two bottoms is less than 5% of the first bottom.
3. The ending point is higher than the local maximum.

```
CREATE PATTERN double_bottom AS
  {downtrend p1;
   uptrend   p2;
   downtrend p3;
   uptrend   p4 WHICH_IS ALL, NON-OVERLAPPING}
   WHICH_IS NON-OVERLAPPING
  ATTRIBUTE date IS last(p4).date
```

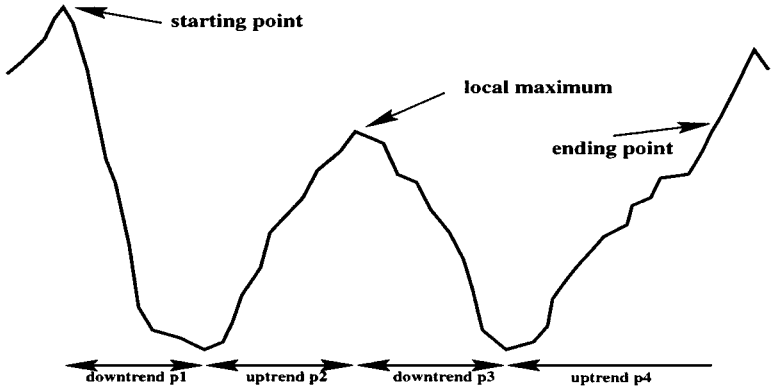

Figure 4. Double-bottom pattern.

```
ATTRIBUTE price IS last(p4).high
WHERE p1.high > p2.high*1.2
      AND abs(p1.low-p3.low) < 0.05*p1.low
      AND p4.high > p2.high
```

Notice that the last date and closing price are kept as attributes. The `FIRST MAXIMAL` search directive of p4 is overridden to `ALL` in order to catch the pattern as soon as possible. Also, the sub-patterns are edge-overlapping because the end of a trend is also the start of the next trend.

2.4. The use of patterns

SQL/LPP+ provides the **BY SEARCHING** clause as an optional quantifier between SELECT clause and FROM clause. The syntax is:

```
BY SEARCHING
   pattern1 alias1 IN time-series_1,
   pattern2 alias2 IN time-series_2,
   ....
   START WHEN start-condition
   STOP WHEN end-condition
   SYNC ON alias1.col_i1,alias2.col_i2,....
```

A query with the `BY SEARCHING` clause collects all patterns found in the corresponding time series. The search progresses from the first element that satisfies the *start-condition*, and stops on the element that satisfies the *end-condition*. When there is more than one pattern under search, the matched segments form multiple streams. The `SYNC ON` clause is used to synchronize these streams and merge them into one.

When `START WHEN` is omitted, the default starting point is the beginning of the time series. When `STOP WHEN` is omitted, the default stopping point is the current end of the time series. When there is more than one pattern in the `BY SEARCHING` clause, the matched

segments of each pattern are aligned by their fields specified in SYNC ON clause. If every pattern finds an instance with an identical value in the fields in the SYNC ON clause, these instances form an element of the answer set.

When patterns are used in the BY SEARCHING clause, only the attributes defined in their definition are accessible, and hence able to be collected into the answers. The content of the segments and local variables are invisible and are discarded after the attributes have been recorded.

The BY SEARCHING clause is placed between the SELECT and FROM clauses. Since usually many matched segments can be found in one time series, the attributes of the matched segments are collected to form a new time series. For example,

```
SELECT ds.symbol,TimeSeries(db.date,db.price)
BY SEARCHING double_bottom db IN ds.quotes
FROM daily_stock ds
```

If a stock has 3 double_bottoms in its time series, the query will report only one record with a time series containing 3 elements.

The query result can also be represented in 1NF format by omitting TimeSeries construction. In that case, each matched segment is a record in the answer.

Time series views are created in a fashion very similar to SQL. A minor difference is that users must name not only table columns but also those columns in time series elements. The following example shows a typical time series view definition.

Example 10. Using the pattern moving_avg_seg defined in Example 8, the following code defines a view which contains the 20-day moving averages of each stock.

```
CREATE VIEW MovAvg20 (symbol20, ma20(date,ma)) AS
  SELECT ds.symbol,TimeSeries(q.date,q.ma)
  BY SEARCHING moving_avg_seg(20) q IN ds.quotes
  FROM daily_stock ds
```

The following example shows how to join the original time series and two derived time series together.

Example 11. The following view contains a time series which contains the closing price, 20 day moving average and 200 day moving average.

```
CREATE VIEW MovAvgs (symbol,
     prices(date,close,ma20,ma200)) AS
  SELECT ds.symbol,TimeSeries(ds.quote.date,
         ds.quote.price,m1.ma,m2.ma)
  BY SEARCHING moving_avg_seg(20) m1 IN ds.quotes
               moving_avg_seg(200) m2 IN ds.quotes
  SYNC ON ds.quotes.date,m1.date,m2.date
  FROM daily_stock ds
```

3. Temporal coupling verification in SQL/LPP+

3.1. Relative relationships of segments

As indicated in Allen (1983), there are 13 possible basic temporal relationships between 2 time series segments. SQL/LPP+ adopts them as the basic relationships used in TCV. To transform these relationships to the form that cascading queries can apply, we define shadow functions.

Definition 16. Given a time series segment $ts[x, y]$ and a temporal relationship R, the *shadow function* Γ is defined as $\Gamma(ts[x, y], R) = \{ts[x', y'] \mid R(ts[x, y], ts[x', y'])\}$.

The shadow functions of the basic temporal relationships are shown in Table 2.

These relationships are used directly in SQL/LPP+ to specify the temporal coupling of two patterns. SQL/LPP+ allows users to combine two or more relationships by the logical connectives `AND` and `OR`. The interpretation of composite relationships is: given a segment s and relationships R_1 and R_2,

$$\Gamma(s, (R_1\ AND\ R_2)) = \Gamma(s, R_1) \cap \Gamma(s, R_2) \quad \text{and}$$
$$\Gamma(s, (R_1\ OR\ R_2)) = \Gamma(s, R_1) \cup \Gamma(s, R_2).$$

Users must be aware that some combinations can result in an empty shadow and should be avoided.

Allen's 13 interval relationships and their combinations can represent any relative temporal relationship. However, an interesting question is how to represent the relationship:

Table 2. The shadow functions of Allen's 13 interval relationships. By default, every relationship has $x \leq y$ and $x' \leq y'$.

#	Relationship (R)	Shadow function $\Gamma(ts[x, y], R)$
(1)	before	$\{ts[x', y'] \mid y < x'\}$
(2)	meets	$\{ts[x', y'] \mid y = x'\}$
(3)	left_overlaps	$\{ts[x', y'] \mid x < x', x' < y < y'\}$
(4)	left_covers	$\{ts[x', y'] \mid x < x', y = y'\}$
(5)	covers	$\{ts[x', y'] \mid x < x', y > y'\}$
(6)	right_covered	$\{ts[x', y'] \mid x = x', y < y'\}$
(7)	equal	$\{ts[x', y'] \mid x = x', y = y'\}$
(8)	right_covers	$\{ts[x', y'] \mid x = x', y > y'\}$
(9)	covered	$\{ts[x', y'] \mid x > x', y < y'\}$
(10)	left_covered	$\{ts[x', y'] \mid x > x', y = y'\}$
(11)	right_overlaps	$\{ts[x', y'] \mid x' < x < y', y > y'\}$
(12)	met	$\{ts[x', y'] \mid x = y'\}$
(13)	after	$\{ts[x', y'] \mid x > y'\}$

pattern A is, at most 10 days and at least 3 days, before pattern B. We introduce *glue*[3] patterns to solve this problem. For example, we can define a glue pattern `glue_3_10` as:

```
CREATE PATTERN glue_3_10 on quote as
  SEGMENT s
  WHERE count(s) >= 3 AND count(s) <=10;
```

Then we define a composite pattern *C* as the concatenation of *A* and `glue_3_10`. The answer to the question above is then that the relationship can be simply represented as `C meets B`. By using glue patterns, users can also specify limits on the length of the overlap of two segments.

3.2. Aggregation and meta-aggregation

A key design goal of SQL/LPP+ is to support summarization of occurrences of pattern coupling. Aggregation on pattern occurrences serves the purpose of summarization. We introduce the syntax and semantics of SQL/LPP+ aggregates in this subsection.

The traditional definition of aggregation is just to find the final result of the aggregates. However, by observing the computation process, we can see that for any aggregate function f and time series t of length n, $f_1(t), \ldots, f_n(t)$ is again a sequence. We can apply aggregate functions on this sequence again and construct an aggregate of aggregates. We call this a *meta-aggregate*.

Take $max(sum(t))$ for an example. We have:

$$max_1(sum_1(t)) = sum_1(t) = t_1$$

$$max_{i+1}(sum_{i+1}(t)) = \begin{cases} max_i(sum_i(t)) & \text{if } max_i(sum_i(t)) > sum_{i+i}(t),\ i+1 \le n \\ sum_{i+1}(t) & \text{otherwise} \end{cases}$$

$$max(sum(t)) = max_n(sum_n(t)) = MAX_{i=1}^{n}\left(\sum_{j=1}^{i} t_j\right)$$

Figure 5 shows an example of $sum(sum(t))$ calculation.

i	1	2	3	4	5	6	7	8	9	10
t_i	3	-2	1	7	-4	-2	1	8	-4	-3
$sum_i(t)$	3	1	2	9	5	3	4	12	8	5
$max_i(sum_i(t))$	3	3	3	9	9	9	9	12	12	12

Figure 5. An example of meta-aggregates $max(sum(t))$ calculation. Given the sequence t shown above. The result of $max(sum(t)) = 12$.

To define meta-aggregates, we start from simple expressions. *Simple expressions* are formed from constants and pattern attributes, and closed under arithmetic operations. For example, assuming p is a pattern alias and c_1 and c_2 are attributes of p, then $p.c_1$, $p.c_1 * p.c_2 + 3$ and $(p.c_1 - p.c_2)/p.c_1$ are simple expressions.

In this paper, we only discuss 5 aggregate functions: *count*, *sum*, *avg*, *min* and *max*. Other aggregates can also be defined in a similar way. Assuming f is an aggregate function, then *aggregate expressions* are defined as:

1. Constants and simple expressions are aggregate expressions.
2. $f(exp)$ is an aggregate expression where *exp* is a simple expression.
3. If E is an aggregate expression, then $f(E)$ is also an aggregate expression.
4. If E_1 and E_2 are aggregate expressions, then $E_1 \odot E_2$ is also an aggregate expression where $\odot$ is an arithmetic operation.

For example, $min(avg(p.c_1))$, $max(p.c_1) - min(p.c_1)$ and $avg(max(p.c_1) - min(p.c_2) + 4) - 2$ are aggregate expressions.

Assuming E is an aggregate expression, we use E_i to denote the aggregate expression that results from replacing every aggregate function f in E with f_i. Assuming $>$, $<$, $+$, $\times$ and $/$ are defined for the corresponding basic type and the number of matched segments is n, the semantics of aggregate functions is defined by:

$$
\begin{aligned}
count_i(E_i) &= i \\
count(E) &= count_n(E_n) \\
sum_1(E_1) &= E_1 \\
sum_{i+1}(E_{i+1}) &= sum_i(E_i) + E_{i+1} \\
sum(E) &= sum_n(E_n) \\
avg_1(E_1) &= E_1 \\
avg_{i+1}(E_{i+1}) &= \frac{avg_i(E_i) \times i + E_{i+1}}{i+1} \\
avg(E) &= avg_n(E_n) \\
max_1(E_1) &= E_1 \\
max_{i+1}(E_{i+1}) &= \begin{cases} max_i(E_i) & \text{if } max_i(E_i) > E_{i+i}, i+1 \le n \\ E_{i+1} & \text{otherwise} \end{cases} \\
max(E) &= max_n(E_n) \\
min_1(E_1) &= E_1 \\
min_{i+1}(E_{i+1}) &= \begin{cases} min_i(E_i) & \text{if } min_i(E_i) < E_{i+i}, i+1 \le n \\ E_{i+1} & \text{otherwise} \end{cases} \\
min(E) &= min_n(E_n)
\end{aligned}
$$

3.3. Single-segment SQL/LPP+ test cases

In rest of this section, we illustrate use of the SQL/LPP+ language with a few examples. First, we demonstrate aggregation and meta-aggregation.

Example 12. Assume a user has constructed a time series view which contains `price` (the daily closing stock price), `ma5` (the 5-day moving average) and `ma20` (the 20-day moving average). Users wants to test a trading strategy called *moving average crossover*: Whenever they see `ma5` cross over `ma20`, they buy 100 shares the next day, holds it till they see `ma20` cross over `ma5` then sell all the holding the next day. They would like to test the strategy and see how well the strategy would have worked on IBM stock. First, we have to create the pattern that represents the period of interest and find the entry and exit price of every trade.

```
CREATE PATTERN crossover ON quote_and_ma AS
  SEGMENT crsovr WHICH_IS MINIMAL, NON_OVERLAPPING
  ATTRIBUTE entry       IS first(crsovr,2).price
  ATTRIBUTE exit        IS last(crsovr,1).price
  WHERE first(crsovr,1).ma5 > first(crsovr,1).ma20
    AND last(crsovr,2).ma5 < last(crsovr,2).ma20
```

The following SQL/LPP+ code creates a *test* that summarizes the performance of this trading strategy. The aggregates we are interested in are the number of trades, the average profit of each trade, the maximal loss in a single trade and the maximal drawdown (the accumulated loss).

```
CREATE TEST crsovr_profit ON quote_and_ma AS
{PATTERN crossover crsovr
 ATTRIBUTE trades       IS count()
 ATTRIBUTE avg_profit   IS avg(crsovr.exit-crsovr.entry)
 ATTRIBUTE max_loss     IS max(crsovr.entry-crsovr.exit)
 ATTRIBUTE max_drawdown IS max(sum(crsovr.entry
                           -crsovr.exit)))
}
REPORT *;
```

Notice `max_drawdown` is defined by the meta-aggregate *max*(*sum*(.)). This test is a single-segment verification. So no temporal relationship is involved. The `REPORT` clause specifies which attributes should be reported. $*$ is a shorthand for all attributes.

The last part of the code specifies the tuple-level SELECT operation which extends SQL by adding a clause

```
BY TESTING test_name test_alias IN time_series_field
```

The main purpose of the following code is to specify which stocks to test and which attributes to report.

```
SELECT qm.symbol, cp.trades, cp.avg_profit,
       cp.max_loss, cp.max_drawdown
BY TESTING crsovr_profit cp IN qm.quotes
FROM quote_and_ma qm
WHERE qm.symbol = "IBM"
```

Processing this SELECT statement first finds the record that contains IBM stock price data, and then searches occurrences of `crossover` while calculating the value of attributes for output.

Suppose there is a daily stock price database which contains quote data spanning 20 years. There are roughly five thousand records and twelve million segments for each stock. In this example 4 quantities, the only things that matter to a trader, are extracted from all the information.

3.4. Multi-segment SQL/LPP+ test cases

In this subsection, we discuss how to define multi-segment tests and use temporal relationships.

Example 13. The following code defines a test to verify whether the famous Double Bottom pattern is really a profitable signal in stock trading.

```
CREATE TEST db_profit ON quote AS
  {PATTERN   double_bottoms db
   ATTRIBUTE db_count        IS count()
   ATTRIBUTE max_entry_price IS max(db.price)
  }
  MEETS
  {SINGLE    PATTERN uptrend ut
   ATTRIBUTE ut_count        IS count()
   ATTRIBUTE avg_profit      IS avg(ut.high-ut.low)
   ATTRIBUTE winning_rate    IS up_count/db_count }
  REPORT winning_rate,max_entry,avg_profit;
```

This segment of SQL/LPP+ defines a test case named `db_profit` on a time series of `quote` type. In the `BY TESTING` clause, each pattern is described by a block delimited by curly braces. The first pattern to be searched for is the `double_bottom` pattern. It is given an alias ‘‘`db`’’. The `ATTRIBUTE` line describes the aggregates to calculate. `count()` keeps the total number of occurrences of this pattern found by the system. `avg(db.price)` calculates the average of the value in the `price` field of every found occurrence. Between the pattern blocks, `MEETS` is the temporal relationship of the patterns to be verified. In the next pattern block, the keyword `SINGLE` denotes that for each occurrence of the preceding pattern, `double_bottom`, the system is to find at most one `uptrend`. (If `MULTIPLE` were placed here, then for each occurrence of `double_bottom`, the system would find as many

uptrends as possible subject to the restart search directives specified in uptrend.) The first pattern does not need the SINGLE/MULTIPLE directive and is always controlled by its own search directives. Each time an occurrence of a pattern is found, only the attributes in the block of that pattern are updated. The REPORT clause lists the attributes to report.

The tuple-level SELECT sentence is omitted because it is very similar to the one in the previous example.

Pattern searching follows the syntactic order in which patterns are specified. When the search engine finds an occurrence of the i-th pattern, it computes the shadow of the segment according to the temporal relationship specified between the i-th and the $(i+1)$-th patterns and attempts to find the occurrence of the $(i+1)$-th pattern. If it cannot find one, it tries to find the next occurrence of the i-th pattern only if MULTIPLE is specified for the i-th pattern.

4. Conclusion

In this paper, we have presented SQL/LPP+. The language provides an intuitive temporal coupling notation to specify combinations of Allen's 13 temporal relationships. We have also extended the concept of aggregation with meta-aggregation. Using meta-aggregation, SQL/LPP+ users can obtain not only the aggregate value of the time series but also the aggregates of aggregates. We believe meta-aggregation is essential in time series data mining.

Currently, temporal coupling verification is a rarely touched area in KDD and data mining. Previous work is isolated in individual application domains. In this paper, we have shown that many problems, in domains from science to stock trading, all have the same problem structure. The contribution of this paper is to provide a level of abstraction for these problems by proposing a language that can formulate the problems of interest. With a TCV language like SQL/LPP+, it becomes possible to simultaneously benefit researchers in numerous fields by improving both the kinds of queries they can ask and the efficiency with which they obtain answers.

Notes

1. If the observation is done at regular time steps, then the result is a *regular* time series. Otherwise, it is *irregular*.
2. See (Perng and Parker, 1999) for extensive discussion of the illustrative and the descriptive approaches.
3. The term *Glue* is borrowed from TEX (Knuth, 1988).

References

Agrawal, R., Psaila, G., Wimmers, E.L., and Zait, M. Querying Shapes of Histories. In *Proc. of the 21st Int'l Conference on Very Large Databases*, Zurich, Switzerland, Sept. 1995.

Allen, J.F. (1983). Maintaining Knowledge About Temporal Intervals. *Communications of the ACM*, 26(11), 832–843.

Berndt, D.J. and Clifford, J. (1996). Finding Patterns in Time Series: A Dynamic Programming Approach. *Advances in Knowledge Discovery and Data Mining*, 229–248.

Edwards, R.D. and Magee, J. (1997). *Technical Analysis of Stock Trends*, 7th edn. AMACOM.

Faloutsos, C., Ranganathan, M., and Manolopoulos, Y. (1994). Fast Subsequence Matching in Time-Series Databases. In *Proc. SIGMOD*.

Fayyad, U.M., Piatetsky-Shapiro, G., Smyth, P., and Uthurusamy, R. (1996). *Advances in Knowledge Discovery and Data Mining*. MIT Press.

Informix Software Inc. (1994) *Informix TimeSeries DataBlade Module User's Guide* Version 3.1.

Knuth, D.E. (1988). *The Texbook : A Complete Guide to Computer Typesetting With Tex*. Addison-Wesley Pub. Co.

Lin, L. and Risch, T. (1998). *Querying Continuous Time Sequences.* VLDB.

Mecca, G. and Bonner, A.J. (1995). *Sequences, Datalog and Transducers*. PODS, 23–35.

Motakis, I. and Zaniolo, C. (1997). Temporal Aggregation in Active Database Rules. *SIGMOD Conference, ACM SIGMOD Record*, 26(2), 440–451.

Perng, C.S. and Parker, D.S. (1999). SQL/LPP: A Time Series Extension of SQL Based on Limited Patience Patterns, DEXA 1999. An extended version is available as Technical Report 980034 UCLA, Computer Science.

Schmidt, D. et al. (1995). Time Series, a Neglected Issue in Temporal Database Research? In J. Clifford and A. Tuzhilin (Ed.), *Recent Advances in Temporal Databases, Workshops in Computing Series* (pp. 214–232). Springer.

Segev, A. and Soshani, A. (1993). A Temporal Data Model Based on Time Sequences. In A.U. Tansel et al. (Eds), *Temporal Databases*. Benjamin/Cummings.

Seshadri, P., Livny, M., and Ramakrishnan, R. (1994). Sequence Query Processing. In *Proceedings of the ACM SIGMOD Conference on Data Management*.

Seshadri, P., Livny, M., and Ramakrishnan, R. (1995). SEQ: A Model for Sequence Databases. In *Proceedings of the IEEE Conference on Data Engineering*.

Shatkay, H. and Zdonik, S.B. (1996). *Approximate Queries and Representations for Large Data Sequences.* ICDE 536–545.

Snodgrass, R.T. (Ed.) (1995). *The TSQL2 Temporal Query Language*. New York: Kluwer Academic Publishing.