# SQL/LPP+: a Cascading Query Language for Temporal Correlation Verification in Time Series Databases

**Chang-Shing Perng**
**perng@cs.ucla.edu**
**University of California**
**Los Angeles, CA 90095-1596**

**Abstract**

Time series data is often generated by continuous sampling or measurement of natural or social phenomena. The resulting quantified records are the basis of scientific analysis and theory construction. In most cases, events are not represented by individual records, and we argue that events can be better represented by time series segments (patterns, temporal intervals). A consequence of this segment-based approach is that the study of temporal correlation among events can be reduced to verifying temporal coupling among occurrences of time series patterns that represent the events.

A major obstacle on the path toward temporal correlation analysis is inability to define interesting time series patterns. We have introduced SQL/LPP[19], which provides fairly strong expressive power for time series pattern query, and are now able to attack the problem of specifying queries that analyze temporal correlation.

In this paper, we propose SQL/LPP+, a temporal correlation verification language for time series databases. Based on the pattern definition language of SQL/LPP, SQL/LPP+ enables users to cascade multiple patterns using one or more of Allen's temporal relationships [1], and obtain the desired aggregates or meta-aggregates of the composition. The issues of pattern composition control are also discussed.

## 1 Introduction

### 1.1 Motivation

Discovering temporal correlation among events is the basis of finding causality of 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 the cause and the effect among events, the simplest and the most intuitive way to deal with the problem is to observe the historical data and check the likelihood that a set of events occur in a particular temporal order. Since the occurrences of events are not predictable, continuous observation or measurement of the subject phenomenon is a common way to record the events. For example, the Richter reading and stock indices need to to be continuously recorded in order to catch important seismic and economic events respectively. The result of these observation and measurement is time series data[1]. Hence time series data is

[1] If the observation is done uniformly periodically, then the result is **regular** time series. Otherwise, it is **irregular**.

usually the only scientific ground to build theories upon. Events, in this representation, are **time series patterns** which possess some special features. For example, the *Great Recession* can be symbolized by the steadily downtrend of Dow Jones Industrial Average(DJIA) in the period of October 29, 1929 to 1932. El Niño and La Niño are recognized by the rising and falling sea water temperature of South America.

We define **Temporal correlation** as the likelihood of different kinds of events occur in a particular time period. To extract knowledge from data, domain experts first propose their hypotheses about the temporal correlation of events, then they have to develop a specialized program to verify their hypotheses. **Temporal correlation verification(TCV)** is the task of finding temporal correlation among events. For example, we would like to know how likely DJIA would decline after a period of high interest rate, and how likely the totally seasonal rainfall in Los Angeles would increase after a 3-month period of high sea water temperature. Even though establishing the real cause and effect relations among these events requires profound domain knowledge, we believe temporal correlation verification is a necessary part of the theory construction and researchers can benefit from utilizing temporal correlation verification languages.

## 1.2 Some Motivating Examples

**Example 1** *This example is a typical case in technical analysis of stocks[14]. A* **Double Bottom** *formation is commonly seen as a pattern that signals a market bottoms. A short-term stock trader might be interested to know what are the winning rate, average profit, maximal loss and total number of trades if he/she had simple bought the stocks when he/she saw the pattern formed and held it for 20 trading days.*

**Example 2** *Every one in the stock market may be interested to know that, in financial history, how likely a downtrend of DJIA can last for more than 6 months during the periods that 30-year T-bond yield was less than* 5.0%.

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

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

By examining these examples, it is easy to see that these problems share some 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 correlation among the events can be verified by counting the occurrences of the corresponding time series patterns. By current available technology, solving each problem requires tedious procedural program coding. Consider the fact 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 is a formal language that can express all these problems, then domain experts no longer need to spend their valuable time in coding/debugging and can concentrate more on perfecting their theory. Computer scientists can also focus on improving the language and the execution efficiency instead of helping domain experts in a case-by-case manner. The urge for a declarative language which provides the ability of fast formulation and execution is obvious.

### 1.3 The Problems

In the aspect of data granularity, knowledge discovery in databases(KDD) and data mining[7, 8] in time series differ from those in set-oriented data. As stated by Shatkay[25],

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

It is continuous time series segments, instead of individual records, that represent events. For example, the stock price of General Electric on someday in 1932, \$8 1/2 a share, does not mean much unless it is compared with the price on September 3, 1929 which was \$396 1/4. Any attempt to design a time series data mining language must provide a way to define patterns which 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 languages. 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 have the flexibility in similarity measurement, the specified patterns might often fail to grasp the real intention of users. We argue that descriptive pattern approach can better express users' ideas of patterns because they allow users to specify the *features* and similarity measurement they want. For example, illustrative patterns can not express concepts like volatility, relative strength and moving averages which are commonly used in financial data analysis[2]. We only discuss descriptive patterns in this paper.

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

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

While occurrences of patterns represent events, it is very hard for a human mind to learn anything from a large number of events. One more 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 every related events. The purpose of TCV languages is to provide that level of abstraction.

[2]See [19] for extensive discussion of the illustrative and the descriptive approaches

Three issues are qraised immediately when attempting to discover the temporal correlation among events.

1. An adequate database model is required to store and manipulate time series data. Time series data is inherently linear ordered. The set-oriented relational model does not capture this characteristic.

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 correlation verification task.

In the next subsection we will discuss the current status in these issues.

### 1.4 Related Work

As indicated in [21], time series databases had been long a neglected issue in database research. Since then, the related research has advanced in many directions. Some data representation models are proposed[22, 23, 24, 11], which addressed the sequentiality of time series data and provided storage models. On the issue of pattern query, the research following the illustrative approach is continuously extending the definition of similarity from exact match[6] to allow some degree of flexibility[4, 13]. On the descriptive approach, TREPL[16], SDL[2] and Sequence Datalog[15] are a few examples of time series related query languages before the introduction of SQL/LPP [19]. To our best knowledge, there is no language specifically designed for knowledge discovery and data mining in time series databases.

### 1.5 The Basic Idea

Recognized the importance of TCV problems, in this paper we attempt to generalize the problem and to design a formal language that is expressive enough to define TCV problems in an unambiguous manner. The language we propose, SQL/LPP+, serves as a problem definition language for TCV problems just like VHDL for hardware description and XML for documentation. Although the LPP model[19] provides a fairly efficient execution model, we do not exclude the possibility that future research can perform better efficiency.

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

1. The patterns of interest, which might be one, two or even more.

2. The temporal coupling relationship(s) among them.

3. The aggregates(statistics) of interest, which might be simple aggregates or meta-aggregates(See 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 many occurrences of the next pattern.

Our idea is to cascade pattern queries. Each pattern, with aggregates associated with it, is specified in a syntactic order and is connected by a combination of temporal relationships. Then the system 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 of defining patterns. So an introduction to SQL/LPP is unavoidable. In Section 2, we discuss some basic issues in designing 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 3.5, we briefly discuss the implementation issues. In Section 4, we conclude with a summary of our current status.

## 2 The SQL/LPP Language

### 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. Since time series is a fairly new data type, we feel the need to introduce 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* $\tau = int \times \tau_1 \times \cdots \times \tau_n$ *where* $n > 0$ *and* $\tau_1, \cdots, \tau_n$ *are basic types. The first field of integer type is the* **index** *of the time series element. The columns of a time series element are named* $c_1, \cdots, c_n$*. The type of time series elements is written as* $\{c_1 : \tau_1, \cdots, c_n : \tau_n\}$.

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

**Definition 3** *A* **time series** *is a function* $ts : integer \rightarrow \mathcal{T}$ *where* $\mathcal{T}$ *is a time series element type and* $index(ts(i)) = i$*, and there exists an integer* $k$*, for every* $i <= k$*,* $ts(i)$ *is defined and for every* $i > k$*,* $ts(i)$ *is undefined.* $ts(i)$ *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 relational model.

**Definition 4** *The* **surrogate** *of a time series is a tuple* $\gamma_1 \times \cdots \times \gamma_m$ *where* $m > 0$ *and* $\gamma_1, \cdots, \gamma_m$ *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, a sequence of time series is a field in a relational table. Surrogates are the keys of tables.

As previously stated, time series segments are the subject to query against. Here is the formal definition and notation of time series segments.

**Definition 6** *A* **time series segment** $s$ *of length* $\ell$ *is a function 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* $ts[k, k + l - 1]$. *The set of all segments is denoted by* $seg(\mathcal{T})$.

The result of a pattern query is a group of segments which can be seen as a *view*. Hence it is important to have a linear ordering on segments in order to keep the same logical structure. The following definitions defines a linear ordering on segments.

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

$\ll: seg(\mathcal{T}) \times seg(\mathcal{T}) \rightarrow$ *boolean 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 resource available and/or the time a requester is willing to wait.*

**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 the language, we can mix the notation of logic sentences and boolean functions.

**Definition 10** *A* **query sentence** $\mathcal{P}$ *is a function 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* $\mathcal{P}(s) = 1$. *It is also denoted as* $\mathcal{P} \models s$.

Even though segments are the subjects of time series queries, it is not practical to use whole segments in data manipulation or output. A few attributes are usually sufficient to represent a segment.

**Definition 12** *An attribute $a$ is a function such that $a : Seg(\mathcal{T}) \rightarrow \tau$ where $\tau$ is a basic type.*

**Definition 13** *An attribute set $Attr$ is a function such that $Attr(s) = (a_1(s), \cdots, a_n(s))$ where $a_1, \cdots, a_n$ are attributes.*

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

**Definition 14** *A* **time series query processing system** *is a function $p : Seg(\mathcal{T}) \times \Phi \rightarrow Seg(Attr(Seg(\mathcal{T})))$ where $\Phi$ is the type of query sentences, such that given a time series $ts$, for every segment $s$ of $ts$, $Attr(s) \in p(ts, P)$ if $P \models s$. Also, $index(Attr(s_1)) < index(Attr(s_2))$ if and only if $s_1 \ll s_2$.*

The query processing only works on a single time series. In order to query multiple time series, we can always merge those time series into a single one. Following definitions define the merge operation.

**Definition 15** *Tuple union operator $\uplus : T_1 \times T_2 \rightarrow T_3$ is defined as $e_3 = e_1 \uplus e_2$ iff $e_1 = (c_{1,1}, \cdots, c_{1,n})$, $e_2 = (c_{2,1}, \cdots, c_{2,m})$ and $e_3 = (c_{1,1}, \cdots, c_{1,n}, c_{2,1}, c_{2,m})$.*

**Definition 16** *A time series $ts$ is the* **merge** *on attribute set $Attr$ of time series $ts_1, \cdots, ts_n$ by synchronizing on fields $c_1, \cdots, c_n$ if $ts$ is the maximal time series such that every element $e$ of $ts$ is $Attr(e_1 \uplus \cdots \uplus e_n)$, where $e_1.c_1 = \cdots = e_n.c_n$ and for any two elements $e'$ and $e''$ of $ts$, $e' = Attr(e'_1 \uplus \cdots \uplus e''_n)$, and $e'' = Attr(e''_1 \uplus \cdots \uplus e''_n)$, $index(e') \leq index(e'')$ if and only if $index(e'_1) \leq index(e''_1), \cdots, index(e'_n) \leq index(e''_n)$.*

The definitions in this subsection lay the necessary groundwork to design time series query languages. The rest of the work is to design the syntax for the languages.

## 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 year to day,
  price  real,
  volume int
 );

CREATE TABLE daily_stocks(
```

| Function | Definition |
|---|---|
| $first$ | $first(s,k) = e$ iff $index_s(e) = k$. |
| $last$ | $last(s,k) = e$ iff $1 \leq \ell - k + 1 \leq \ell$ 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 $s$, $e.c_i \leq u$. |
| $min$ | $min(s,c_i) = v$ iff there exists an element $e$ in $s$ such that $e.c_i = v$ and for all $e$ in $s$, $v \leq e.c_i$. |
| $is_first$ | $is_first(e,k) = T$ iff $1 \leq k \leq \ell$ 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 $index_s(e_1) = m$, $m + k \leq \ell$ and $index_s(e_2) = m + k$. |
| $previous$ | $previous(e_1,k) = e_2$ iff $index_s(e_1) = m$, $m - k \geq 1$ and $index_s(e_2) = m - k$. |

Table 1: SQL/LPP Functions

```
  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 decided by an indexing function which, in the real world, 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.

## 2.3 Defining SQL/LPP Pattern

### 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 expandable. Users can add more functions to the language if they feel it is necessary.

| 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 1: The segments selected by different selection search directives are MINIMAL:$ts[1,5]$, FIRST MAXIMAL:$ts[1,7]$ and MAXIMAL:$ts[1,9]$. The truth value is evaluated against the pattern sentence $first(s).price < last(s).price$.

### 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:

- $is_first(e)$,
- $last(s,3).price > 100$,
- $avg(s, volume) * count(s) = 3141000$,
- $next(first(s,3),3).low - previous(last(s,2)).high > last(s).price$ and
- $last(s).price > e.price$.

**Pattern formulae** are defined inductively:

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

A pattern formula $\mathcal{P}$ is a **pattern sentence** if every element variable occurrences in $\mathcal{P}$ are 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 must be always assigned **F**. 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 $ts[x,y] \models (NOT\ previous(first(s)).price > 0)$.

### 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 with another. Consider an **uptrend** pattern in a daily stock price database is a continuous period which satisfies the following two conditions:

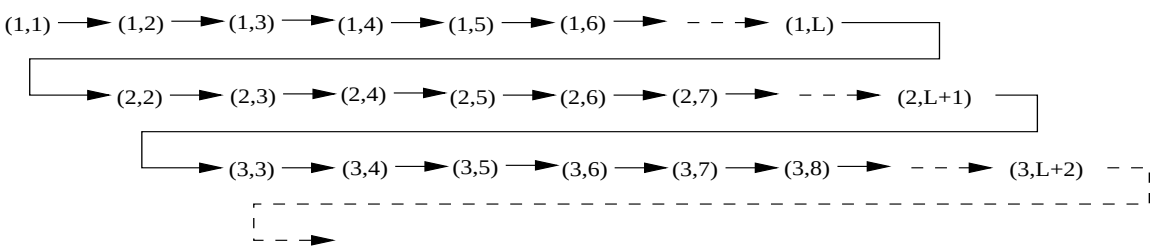



Figure 2: The segment space ordered by lexicographical ordering

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 which 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 must provide search directives to reduce the number of matched segments.

Based on the segment ordering shown in Figure 2, 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.

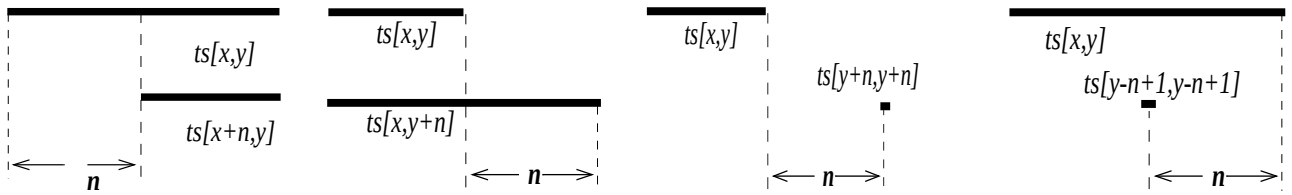
Assume the description of a pattern is a pattern sentence $\mathcal{P}$. The following are SQL/LPP selection search directives:

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 the longest 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.
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 1, 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.




(a) SKIP START $n$ (b) SKIP END $n$ (c) SKIP $n$ (d) MAX-OVERLAPPING $n$

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 $(1, 10)$ 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. The syntax of pattern definition is presented in this subsection.

### 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 >= previous(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 provide 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). 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}
```

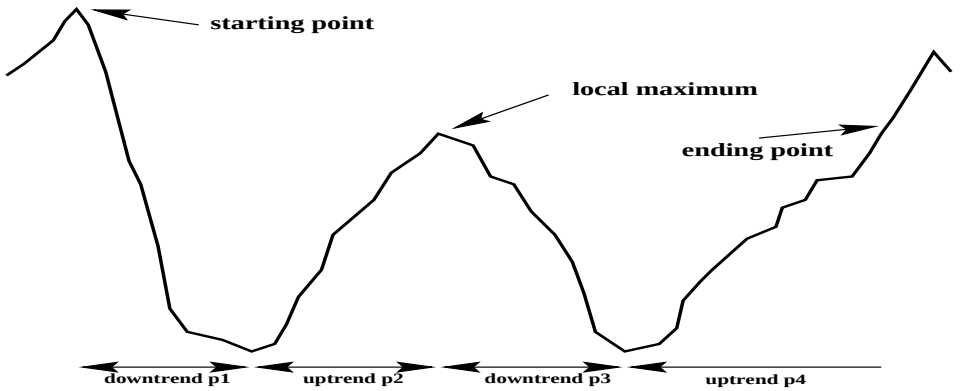



Figure 4: Double-bottom pattern

Edge-overlapping composition is same as the case of 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
  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 are 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 *BY SEARCHING* clause as an optional part of queries. 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 identical value in the fields in `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, 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 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
```

As in SQL, SQL/LPP views can be queried as regular tables. Patterns defined on views must specify the type of the time series element. For example, given the definition of view `MovAvg20` in Example 10, a row type `type_of(MovAvg.ma20)` is defined by:

```
CREATE ROW TYPE type_of(MovAvg20.ma20)(
  date  datetime year to day,
  ma    real
)
```

The column names are taken from the view definition and the types of the columns are inferred by propagating the types in the original table. Other patterns can be defined on the derived type without defining a row type and risking name or type inconsistency.

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 The SQL/LPP+ Language

## 3.1 Relative Relationships of Segments

As indicated in [1], given 2 time series segments, there are 13 basic temporal relationships if we include all symmetric cases. SQL/LPP+ adopts them as the basic relationships in TCV problem.

To transform these relationships to the form that cascading querying can apply, we define the shadow functions.

**Definition 17** *Given a time series segment* $ts[x, y]$ *and a temporal relationship* $R$*, the* **shadow function** $\Gamma$ *is defined as* $\Gamma(ts[x, y], R) = \{ts[x', y']|R(ts[x, y], ts[x', y'])\}$.

The shadow functions of 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 logical connectives `AND` and `OR`. The interpretation of composite relationships is: given a segment $s$ and relationships $R_1$ and $R_2$,
$\Gamma(s, [R_1\ AND\ R_2]) = \Gamma(s, R_1) \cap \Gamma(s, R_2)$ 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 empty shadow and should be avoided.

Allen's 13 interval relationships and their combinations can represent any relative temporal relationships. However, an interesting question is how to represent the relationship: *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 relationship mentioned above can be simply represented as `C meets B`. By using glue patterns, users can also specify the length range of the overlapping part of two segments.

### 3.2 Aggregation and Meta-Aggregation

The purpose of designing SQL/LPP+ is to summarize the occurrences of interesting pattern coupling. Aggregation on pattern occurrences serves the purpose of summarization. We introduce the syntax and semantics of SQL/LPP+ aggregates in this subsection.

We first take aggregate functions *sum* and *max* as examples. Given a sequence $t = [t_1, \cdots, t_n]$, where $t_1, \cdots, t_n$ are of type $\tau$ and $>$(larger than) and $+$(plus) operators are defined on $\tau$, then

$$\begin{aligned} sum_1(t) &= t1 \\ sum_{i+1}(t) &= sum_i(t) + t_{i+1} \\ sum(t) &= sum_n(t) \\ max_1(t) &= t1 \end{aligned}$$

[3]The term *Glue* is borrowed from TEX[12].

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

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

$$\begin{aligned} max_{i+1}(t) &= \begin{cases} max_i(t) & \text{if } max_i(t) > t_{i+i},\ i+1 \leq n \\ t_{i+1} & \text{otherwise} \end{cases} \\ max(t) &= max_n(t) \end{aligned}$$

The traditional definition of aggregation is the final result of the aggregates. However, by observing the computation process, we can see that for any aggregate function $f$, $f_1(t), \cdots, f_n(t)$ is again a sequence. The sequence is the collection of the result of the aggregate in some particular time point. It is itself valuable information. So we can apply aggregate functions on this sequence again and construct an aggregate of aggregates. We call it **meta-aggregate**. Take $max(sum(t))$ for an example. We have:

$$\begin{aligned} max_1(sum_1(t)) &= sum_1(t) = t_1 \\ max_{i+1}(sum_{i+1}(t)) &= \begin{cases} max_i(sum_i(t)) & \text{if } max_i(sum_i(t)) > sum_{i+i}(t),\ i+1 \leq n \\ sum_{i+1}(t) & \text{otherwise} \end{cases} \\ max(sum(t)) &= max_n(sum_n(t)) = MAX_{i=1}^{n} \left( \sum_{j=1}^{i} t_j \right) \end{aligned}$$

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

Meta-aggregates can not be constructed arbitrarily. For example, $max(t)+t$ is not a valid aggregate expression. To define meta-aggregates, we start from simple expressions. **Simple expressions** are formed by constants and pattern attributes, and complete under arithmetic operations. For

| $i$ | 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 $max(sum(t)) = 12$.

example, assume $p$ 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 the similar way. Assume $exp$ is a simple expression, $f$ and $f'$ are aggregate functions, **aggregate expressions** are defined as:

1. Constants and $f(exp)$ are aggregate expressions.
2. If $E$ is an aggregate expression, then $f'(E)$ is also an aggregate expression.
3. 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 but $p.c_1$ and $avg(p.c_1) + p.c_2$ are not.

Assume $E$ is an aggregate expression, we use $E_i$ to denote the aggregate expression resulted by replacing every aggregate function $f$ in $E$ with $f_i$. Assume $>$,$<$,$+$,$\times$ and $/$ are defined in the corresponding basic type, the semantics of aggregate functions is defined as:

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

$$
\begin{aligned}
max(E) &= max_n(E_n) \\
min_1(E_1) &= E_1 \\
min_{i+1}(E_{i+1}) &= \begin{cases} min_i(E_i) & \text{if } min_i(E_i) < E_{i+i},\ i+1 \leq n \\ E_{i+1} & \text{otherwise} \end{cases} \\
min(E) &= min_n(E_n)
\end{aligned}
$$

Note that the value of *count* does not depend on its argument. We use *count*() to denote any $count(EXP)$ where $EXP$ is a valid argument.

### 3.3 Single-segment SQL/LPP+ Test Cases

In rest of this section, we present the SQL/LPP+ language by a few examples. First, we demonstrate the aggregation and mega-aggregation of SQL/LPP+.

**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). The user wants to test a trading strategy called* **moving average crossover***: Whenever he/she sees* `ma5` *cross over* `ma20`*, he/she buys* 100 *shares the next day, holds it till he/she sees* `ma20` *cross over* `ma5` *then sells all the holding the next day. He/She would like to test the strategy and see how well whould have the strategy worked on IBM stock. First, we have to create the pattern which 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 create 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_single_loss   IS max(crsovr.entry-crsovr.exit)
    ATTRIBUTE max_drawdown      IS max(sum(crsovr.entry-crsovr.exit)))
  }
  REPORT *;
```

*Note that* `max_drawdown`*, is defined by 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 what stocks to test and what attributes to report.*

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

When the SELECT statement is issued to a SQL/LPP+ system, the system will first find the record that contains IBM stock price data, then search the occurrences of `crossover` and calculate the value of attributes for output.

Suppose there is a daily stock price database which contains data of 20 years. There are roughly 5000 records and $12,502,500$ segments for each stock. This example is a demonstration of extracting only 4 quantities, the only things matter to a trader, from all the information.

### 3.4 Multi-segment SQL/LPP+ Test Cases

In this subsection, we discuss how to define multi-segment tests in SQL/LPP+ and the use of 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 time series of* `quote` *type. In* `BY TESTING` *clause, each pattern is described in a block formed by* { *and* }*. The first pattern to search 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* `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_bottoms`*, the system finds at most one* `uptrend`*. If* `MULTIPLE` *is placed here, then for each occurrence of* `double_bottoms`*, the system finds as many* `uptrend` *as possible while subjected to the restart search directives specified in* `uptrend`*. The first pattern does not need* `SINGLE/MULTIPLE` *directive and is always controlled by its own search directives. An analogy of* `SINGLE` *is the* cut(!) *symbol in Prolog Language. 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.*

The procedure semantics of SQL/LPP+ test cases is:

Function find(i,shadow)
BEGIN
  WHILE not end of shadow
    If found a $P_i$'s occurrence $s$ then
      Update aggregates in $P_i$ block;
      If $P_{i+1}$ exists
        call find$(i+1, shadow(s, R_{i,i+1}))$;
      If `SINGLE` is specified for $P_i$ , return;
    Move search cursor according to restart directives;
  End While;
END Find;

Algorithm TCV
BEGIN
  Input Patterns $P_1, \cdots, P_n$ ;
    Temporal Relationships $R_{1,2}, \cdots, R_{n-1,n}$;
    Time Series $ts$;
  Call find$(1, ts)$;

Report aggregates;
END TCV;

### 3.5 Implementation

An efficient implementation of SQL/LPP+ engine depends on how fast a system can achieve the following tasks:

1. Find segments which match patterns in a particular interval.
2. Update attributes.
3. Compute the shadow area.

It is easy to see that the last two tasks are trivial comparing to pattern searching. So the main issue in optimization SQL/LPP+ queries is the same as in SQL/LPP. In [19], we proposed an execution model called **Limited Patience Pattern(LPP)**. Although the number of segments is $O(n^2)$ in a length $n$ time series, most attributes of a segment can be computed from the attributes value of its neighboring segments and the difference of the two segments. This fact is the basis of **incremental computation** in LPP. For example, in a time series of length $n$, computing $m$ day moving average has complexity $O((n-m+1)\times m) = O(nm)$. With incremental computation, the complexity can be improved to $O(m+(n-m+1)) = O(n)$. We have similar speed-up for other aggregates, median and even the Fourier transformation. LPP model uses attributed queues as the basic data structure for incremental computation.

Also, by analyzing the syntactic structure of SQL/LPP pattern definition, we can find the upperbound and the lowerbound of the length of the satisfying segments. In some cases, the distance between two length bounds can be very narrow and the search space can be effectively reduced. Also, there are many types of invariabilities associated with query sentences. For example, if a segment $ts[x,y]$ does not satisfy the query sentence `[ALL e IN s](is_first(e,1) OR e.price>previous(e).price)` of `up_trend`, then we can conclude that none of the segments $ts[x,z]$, where $z > y$, can satisfy the query sentence. Hence there is no need to evaluate these segments. Six invariabilities are discussed in [19].

## 4 Conclusion

In this paper, we present SQL/LPP+. The language is based on the pattern defining and searching ability of SQL/LPP. SQL/LPP+ provides an intuitive temporal coupling notation to specify single or a combination of Allen's 13 temporal relationships. We also extended the concept of aggregation and introduced meta-aggregation. With meta-aggregation, SQL/LPP+ users can obtain not only the aggregate value of the time series but also the aggregates of aggregates. We believe it is essential in time series data mining.

Currently, temporal correlation verification is a rarely touched area in KDD and data mining. Previous work is isolated in each domain. In this paper, we showed that many problems, in domains from natural science and social science to stock trading, all have the same problem structure. The contribution of this paper is to provide a level of abstraction on 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 the language and the efficiency of test case evaluation.

## References

[1] Allen, J.F. *Maintaining knowledge about temporal intervals.* Communications of the ACM, vol.26, (no.11), Nov. 1983. p.832-43.

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

[3] R. Agrawal, R. Srikant. *Mining Sequential Patterns*, Proc. of the Int'l Conference on Data Engineering (ICDE), Taipei, Taiwan, March 1995.

[4] D. J. Berndt, J. Clifford. *Finding Patterns in Time Series: A Dynamic Programming Approach.* Advances in Knowledge Discovery and Data Mining 1996: 229-248

[5] Surajit Chaudhuri, Umeshwar Dayal, "An Overview of Data Warehousing and OLAP Technology", *IEEE Data Engineering*, December 1997.

[6] C. Faloutsos, M. Ranganathan, Y.Manolopoulos. "Fast Subsequence Matching in Time-Series Databases", *Proc. SIGMOD'94*.

[7] Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, *Advances in Knowledge Discovery and Data Mining*, ISBN 0-262-56097-6, MIT Press, 1996.

[8] U.M. Fayyad, "Mining Databases: Towards Algorithms for Knowledge Discovery", *IEEE Data Engineering*, March 1998.

[9] E. F. Codd. *A Relational Model for Large Shared Data Banks* Communications of the ACM, Vol.13 No.6, June 1970.

[10] E. Gamma, et al. Design Patterns : Elements of Reusable Object-Oriented Software Addison-Wesley Professional Computing, 1995.

[11] Informix Software Inc. *Informix TimeSeries DataBlade Module User's Guide* Version 3.1.

[12] D.E. Knuth. *The Texbook : A Complete Guide to Computer Typesetting With Tex.* Addison-Wesley Pub Co; ISBN: 0201134489 , April 1988.

[13] Ling Lin, Tore Risch. *Querying Continuous Time Sequences.* VLDB 1998.

[14] R. D. Edwards, J. Magee. *Technical Analysis of Stock Trends.* 7th Edition, AMACOM, May 1, 1997.

[15] G. Mecca, A. J. Bonner. *Sequences, Datalog and Transducers*, PODS 1995: 23-35

[16] I. Motakis, C. Zaniolo. *Temporal Aggregation in Active Database Rules* SIGMOD Conference 1997: 440-451

[17] D. S. Parker *Stream Data Analysis* appeared in: L. Sterling, ed., The Practice of Prolog, Cambridge, MA: MIT Press, 1990.

[18] D.S. Parker, E. Simon, P. Valduriez *SVP – A Model Capturing Sets, Streams, and Parallelism* Proc. 18th Conf. on Very Large Data Bases, Vancouver, British Columbia, August 1992.

[19] C.S. Perng, D.S.Parker SQL/LPP:a Time Series Extension of SQL based on Limited Patience Patterns, Technical Report 980034 UCLA, Computer Science.

[20] W. Dreyer, A. K. Dittrich, and D. Schmidt, *Research Perspectives for Time Series Management Systems*, ACM SIGMOD Record, Vol. 23, No.1, 1994

[21] D. Schmidt et al. *Time Series, a Neglected Issue in Temporal Database Research?* In J. Clifford and A. Tuzhilin, editors, Recent Advances in Temporal Databases, Workshops in Computing Series, pages 214-232. Springer, 1995.

[22] A. Segev and A. Soshani, *A Temporal Data Model Based on Time Sequences*, in Temporal Databases, A.U. Tansel et al. (eds), Benjamin/Cummings, 1993.

[23] P. Seshadri, M. Livny, R. Ramakrishnan. *SEQ: A Model for Sequence Databases.* Proceedings of the IEEE Conference on Data Engineering, March 1995.

[24] P. Seshadri, M. Livny, R. Ramakrishnan.*Sequence Query Processing.* Proceedings of the ACM SIGMOD Conference on Data Management,May 1994.

[25] Hagit Shatkay, Stanley B. Zdonik. *Approximate Queries and Representations for Large Data Sequences.* ICDE 1996: 536-545