SQL/LPP: a Time Series Extension of SQL based on Limited Patience Patterns

Chang-Shing Perng **D. Stott Parker**
perng@cs.ucla.edu stott@cs.ucla.edu
Computer Science Department
University of California
Los Angeles, CA 90095-1596

Abstract

Time series data is essential in almost every kind of natural and social science research. It is also the linchpin of information used in economical and political decision making. Surprisingly, however, today we still lack an adequate time series database model and an expressive time series query language.

The significance of this lack is underscored by the strong recent trend toward incorporating greater analytical power in data management systems (witnessed by the amazing growth of interest in OLAP and KDD). This trend reflects the current reality of ubiquitous information and the importance of interpreting it rapidly.

Time Series Databases have proven more difficult to treat with known methods than other types of databases, and the currently available tools are insufficient to analyze and explain this special type of data. Time series databases differ from relational databases in two aspects. First, the sequentiality of data does not fit in the set-oriented relational database model. Second, the subjects of time series queries are more likely to be segments (temporal intervals) of the time series than temporally uncorrelated records, and querying time series is more like pattern matching than simple retrieval.

In this paper, we introduce SQL/LPP, a time series extension of SQL. SQL/LPP is based on Limited Patience Patterns, a temporal pattern model that is expressive, practical, intuitive, and supports a declarative algebraic syntax permitting query optimization. Limited Patience Patterns can be interpreted in multiple equivalent ways: patterns definable on bounded-length time series segments, patterns composed from limited-memory transducers (parsers), concatenable queues of recognizers, or intensional time series whose types are inductively defined by simple grammars. We illustrate basic features of SQL/LPP, showing the definition and use of popular time series patterns. We also show how the definition of SQL/LPP permits both static and dynamic analysis, permitting evaluation of the complexity of queries and optimization for efficient evaluation whenever it is possible.

1 Introduction

1.1 Motivation and the Problem

Time series data is being recorded today at an accelerating pace. In numerous fields, from economic indices and financial market information to various scientific instrument readings, the daily activity of decision makers is to analyze time series and to uncover their implications. This phenomenon is part of a global trend towards more extensive tracking of information and more powerful analysis of what it means. The explosive recent growth of interest in OLAP (On-Line Analytical Processing), [3] and KDD (Knowledge Discovery in Databases) [5, 6] reflects the current confluence of factors (globalization, increased competition, reduced cost of storage, Moore's law, etc.) that together have pressed for much greater analytical power in data management systems.

Despite the obvious need for systems to manage time series data, such systems have not been developed by the database research community, as noted in [17, 16]. This is perhaps best explained by the two key differences between the time series and relational perspectives. First, the *sequentiality of time series* does not fit well into the set-oriented relational model[7]. (As a result, many alternative models[18, 19, 9] have been proposed to preserve the temporal ordering among records.) Second, in time series *continuous segments* are the subjects of queries — instead of discrete, temporally uncorrelated records — and querying a time series database is actually more like pattern matching than performing relational algebra operations.

The differences here are quickly highlighted by simple examples. A typical time series application area is searching for patterns in technical analysis of stock charts[10]:

Example 1 *An* **uptrend** *pattern in a daily stock price database is a continuous period which 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.*

SQL3, the current de facto standard query language, does not respond well to challenges like this. The well-known temporal query language TSQL2[21] does not have the expressive power to define general patterns either.[1] There is a serious lack of adequate languages for expressing time series queries.

Another reason for proposing a query language for time series pattern query is to lay the groundwork for *temporal correlation verification*(TCV)[15]. By TCV, we mean to verify how likely some events always occur along with other events. For example,

- Economists might want to know how many bear markets, when DJIA falls 20% from its previous high, were preceded by a period of high interest rate.
- Seismologists might be interest to know whether major earthquakes were always preceded by a period with a number of smaller earthquakes.
- A stock trader might want to know if he/she had bought a stock whenever he/she saw a Double Bottom formation then held it for 20 days, would he/she be profitable.

[1]TSQL2 does include a `RISING` aggregate, which is similar in ways to the notion of an uptrend here. However, TSQL2 generally lacks pattern constructs, and was not designed for time series analysis.

To deal with these TCV problems, a language capable of defining patterns and synthesizing time series is required. TCV, to our best knowledge, is currently a rarely touched field in KDD and data mining. By examining the importance of the above problems, we believe TCV will not be ignored forever.

With the foregoing in mind, this paper is organized in the following way. In Section 2, we present the basic idea behind all the work in this paper. We discuss in detail the pros and cons of two pattern definition approaches, illustrative and descriptive. We model the problem to mathematical terms and show it is possible to design a language that unifies both approaches. Then we describe Limited Patience Patterns model for time series. This model gives an intensional foundation for time series databases that tries to balance complexity restrictions with the goals of expressiveness and intelligibility. In Section 3, we introduce SQL/LPP, a time-series extension of SQL and is a basic implementation of the Limited Patience Patterns model. This implementation can be extended in many ways, but it illustrates the expressive power of the model. In Section 4, we propose a static analysis technique of the syntax structure of SQL/LPP queries and identify the properties they possess. Then we discuss the dynamic analysis technique of using both the static properties of the query and evaluation-time information to direct heuristic search in the search space. In Section 6, we conclude with a summary of our current status and the future work.

1.2 Previous Work

Several languages have been proposed to serve as a query language for 'time series' databases or similar purposes. Examples include the following:

- *TREPL*[12] is a composite event description language which provides strong expressive power in describing time series patterns. A fundamental difference between TREPL and a conventional query language is that TREPL inspects database transaction event patterns instead of just data patterns; in cases where transaction time and real time differ, TREPL can produce counterintuitive results. TREPL does not provide an incremental execution model for efficient implementation, which we believe is critical for time series applications.

- *SVP*[14] is a data model that represents both streams and sets as trees, which can be queries using limited functional mappings called transducers. The transducer limitations significantly restrict the expressive power of queries, making it infeasible to express concepts like 'median'.

- The Informix TimeSeries DataBlade [9] provides time series types for storage management. It does not extend the SQL language to handle the records in time series tables. Instead, it defines hundreds of time series functions to express queries. Because time series records are accessible only through these functions, it is difficult to develop new functions and this approach is not truly extensible.

- *SEQUIN* [19][20] extends SQL to allow efficient evaluation when database records are sequentially ordered. However, SEQUIN is comparatively weak in expressing pattern queries.

- *SDL*[1] is a shape definition language. It is aimed at numerical (single-quantity) time series.

- *Sequence Datalog* [11] is an extremely powerful language (a type-0 language in the Chomsky Hierarchy), hence termination of Sequence Datalog queries is not guaranteed. It was originally designed for Genome databases, so the temporal ordering is not taken into account for efficient evaluation, and it lacks an execution model with reasonable complexity.

- Berndt and Clifford [2] proposed a dynamic-programming approach to find time series pattern, but they did not propose a formal language to define patterns. Also, their approach has very high time complexity.

2 A Unified Pattern Searching Framework

In this section we propose an execution model called **Limited Patience Patterns**. A central idea of Limited Patience Patterns is the use of **attributed queues**. An attributed queue is a queue of variable but limited size (hence 'limited patience'), with some defined attributes which are updated whenever an insertion or a deletion operation is performed.

The incremental evaluation model is effective in practical querying of time series. For one thing, the incremental evaluation model conforms to most commonly used time series analysis models which are incrementally computable, including moving averages, autoregressions, ARIMA, etc. Put another way, attributed queues (or Limited Patience Patterns) are good at expressing analytical queries with computational efficiency.

The data organization model is the one already in use by commercial products like the Informix TimeSeries DataBlade[9]. The basic idea is to use non-first-normal-form (N1NF) objects to represent time series. A time series consists of two parts: the **surrogate** and the **body**. The surrogate serves as the key of the table. The body is a sequence of **elements** of a user-defined row type. For example, in stock price database the stock names and trading symbols are the surrogates because they are invariable over time. The market information of the stocks are the bodies.

Beyond these things, however, Limited Patience Patterns seems to provide a satisfactory way to resolve basic issues in managing time series data.

2.1 Basic Issues in Time Series Data Management

2.1.1 Illustrative Patterns vs. Descriptive Patterns

The first issue in time series pattern query is to find an appropriate way to express queries. There are two main approaches in expressing time series patterns:

- **Illustrative Approach**: A typical illustrative pattern query issuing procedure is to draw a chart or select a segment from an existing library as the pattern, then send the chart to a query processing engine to find similar segments.

- **Descriptive Approach**: Query issuers express the pattern in a formal language provided by the database system.

There seems to be very little, if any, discussion and comparison of the pros and cons of these two approaches, or whether they can be unified somehow.

Time series query languages thus spread on the spectrum of complexity and expressive power. On the illustrative end, queries are expressed purely in a "just-like-this" fashion [2, 4] that requires no real language at all. Illustrative patterns, obviously, are easier to specify and use. However the expressive power of this approach is usually limited because all graphical patterns are interpreted in an inflexible system-built-in way.

On the descriptive end of the spectrum, Sequence Datalog[11] and general purpose programming languages can express all Turing machine-recognizable patterns. The expense of the approach is its lack of termination guarantee, greater difficulty of use, and potentially intractable complexity.

We claim that either extreme is inadequate. On one hand, decades of effort in developing query languages has made evident why declarative query languages are better than programming languages for representing queries by relieving the programming burden of database application programmers. On the other hand, the graphical approach and transducer-based languages are poor in expressive power.

2.1.2 Approximate vs. Exact Retrieval

In many situations, querying a time series database is often more like pattern matching than simple relational retrieval. In such situations, it can make more sense for patterns to be matched **approximately** instead of exactly. So the key issue is how to measure the **similarity** between two time series segments.

This is a difficult issue to resolve. For example, in [4] and [2], the only measurement is Euclidean distance. Although we consider Euclidean distance generally to be a good measurement, it is certainly not adequate in every case, especially when query issuers have different ideas about the characteristics of interesting patterns.

For example, a stock trader may want to find periods that are longer than ten trading sessions and during which a stock went up every day. The trader may draw a picture like fig 1(a) which he/she thinks is good enough to express the idea of *monotonic increase*. However, the outcome might be quite surprising based on the Euclidean distance as shown in fig 1(b)-(d). If the illustrative pattern searching engine is capable of perceiving similarity of other definitions, it may still fail to capture the real intent of the user.

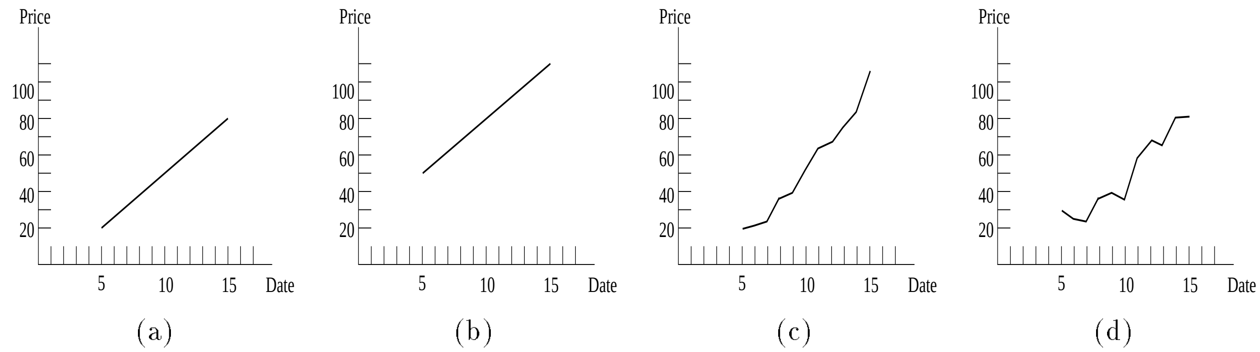


Figure 1: Suppose **monotonic uptrend** is the pattern in mind when a user draws (a) as the example. He/She might think (b) and (c) are qualified answers and (d) is not. However, when similarity is defined in terms of the Euclidean Distance, the result is exactly the opposite. (b) has exactly the same length and shape as those of (a) but of different height, hence the Euclidean distance between (a) and (b) is quite large. (c) satisfies the monotonic increasing conditions but the difference between the slopes of (a) and (c) causes a large Euclidean distance. (d) is not monotonic increasing but it is the most 'similar' to (a).

So, the issue here is that on one hand, a system must provide as many ways of measuring similarity as possible to enhance its expressive power, or the system will be too primitive to express even very basic patterns. On the other hand, a system capable of defining various kinds of similarity may put the wrong emphasis into measurement. The issue becomes very similar to that of illustrative vs. descriptive pattern specification.

The way Limited Patience Patterns deal with approximate matching is by converting the problem to exact matching in a very straightforward way. For example, finding segments with feature value approximate to v with error less than ϵ is equivalent to finding segments with feature value inside $[v-\epsilon, v+\epsilon]$. Hence the distance measurement problem is converted to a true-or-false problem and still has the same meaning.

2.1.3 Complexity

While designing new query languages, it is crucial to study the nature of the underlying problem. That is, the language design must be based on a working execution model which is sound and efficient. Time series pattern searching is more expensive than the selection operations of relational algebra. In a time series of length n, there are $n(n+1)/2$ segments, and the search space is of size $O(n^2)$. As time series are usually very long (n is large), no time series query language can succeed without a well-optimized execution model. Thus time series and relational databases also differ significantly over computational complexity issues.

From a computational perspective, the expressive power of various languages is differentiated by the way their execution models access data. Turing machine-equivalent languages can randomly access data an unlimited number of times. Transducer-based languages read data only once and accordingly change their fixed set of state variables. A key issue is to identify an effective execution model that will limit the scope of complexity issues to be faced by the query optimizer.

2.1.4 Time Series Segments vs. Individual Elements

A final issue is the granularity on which to base the query language: time series segments (subintervals), or time series elements. This issue has been the inspiration for religious battles over almost all computer languages with a temporal aspect, and it appears it has no definitive general resolution.

However, in the time series context, there does seem to be a resolution. In the typical case where time series are primarily just sequentially sampled values of time-varying features, segments appear to offer advantages as the subjects of queries on time series than do individual elements. Segments allow natural expression of 'state' concepts which are relevant here, and segments include individual elements as a degenerate special case (while the reverse does not hold). Furthermore segments are the natural dual of patterns: a pattern *matches* a segment of a time series. Let us formalize these ideas.

Definition 1 *A* **time series element** *is an object of type* $\mathcal{T} = \tau_1 \times \cdots \times \tau_n$ *where* $n > 0$ *and* $\tau_1, \cdots, \tau_n$ *are basic types. The columns of a time series element are named* $c_1, \cdots, c_n$. *The type of time series elements is written* $\{c_1 : \tau_1, \cdots, c_n : \tau_n\}$.

Definition 2 *A* **time series** *is a function* $ts : integer \rightarrow \mathcal{T}$ *where* $\mathcal{T}$ *is a row type.* $ts(i)$ *is usually denoted by* $ts[i]$.

Definition 3 *Given a time series* ts *of type* $\mathcal{T}$, *a function* s *is a* **time series segment** *of length* ℓ *if there exists a integer* $k > 0$ *such that for all* $1 \leq i \leq \ell$, $s[i] = ts[k+i-1]$. *The set of all segments is denoted by* $seg(\mathcal{T})$.

In the rest of this paper, we simply use **elements** and **segments** for time series elements and time series segments when the context is clear.

Given a time series ts, a segment s can be represented as $ts[u, v]$ which indicates indices of the first and the last elements. That is, $s[1] = ts[u]$ and $s[\ell] = ts[v]$ where s is of length ℓ. We assume the length of a segment is at least 1.

For each time series element $s[i]$, its index is simply the position of the time series element in the time series. That is, given s as a time series or segment, $s[i]$ is an element in s, and $index(s[i]) = i$.

Now we define the natural ordering of segments. The **segment ordering** is defined as the pairwise lexicographic ordering $\ll$: $seg(\mathcal{T}) \times seg(\mathcal{T}) \rightarrow boolean$ such that

$$ts[x_1, y_1] \ll ts[x_2, y_2] \iff (x_1 < x_2) \vee (x_1 = x_2 \wedge y_1 < y_2)$$

It is well-known that the (lexicographic) segment ordering defines a linear ordering in $\mathcal{S}_L$. Among other things, this is significant in that it gives a preference ordering among all qualified

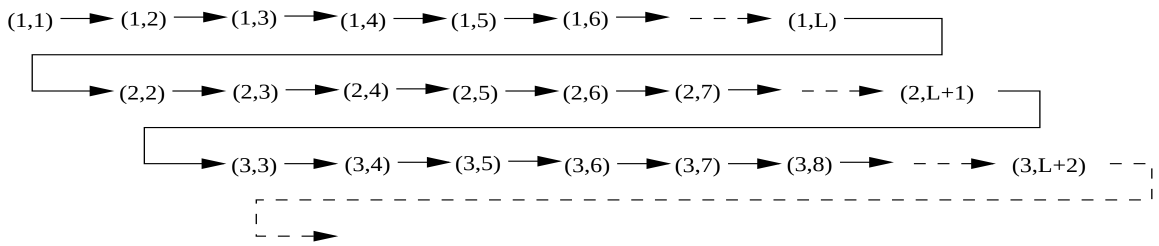
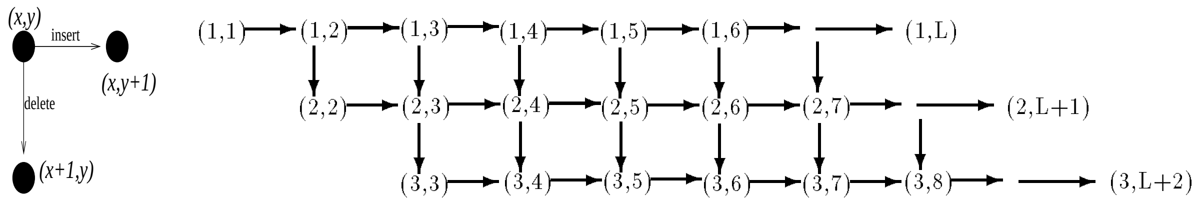


Figure 2: The segment space ordered by lexicographical ordering



(a) Transitions (b) The Search Space

Figure 3: The segment space as a traversal graph. The length limit is L. A segment changes to the one on its right when insertion is performed. It changes to the one below itself when an element is deleted.

answers to a query. Also, the lexicographic ordering detects emerging patterns earlier instead of later, and on length-limited segments it is a linear ordering — which is very useful, as will be discussed in Section 3.4. With the segment ordering, a successor function $succ : seg(\mathcal{T}) \rightarrow seg(\mathcal{T})$ is defined as: $succ(s_1) = s_2$ if $s_1 \ll s_2$ and for all s_3, if $s_1 \ll s_3$ then $s_2 \ll s_3$ or $s_2 = s_3$.

2.2 Limited Patience Patterns

By examining Figure 2, we can see that the adjacent nodes(segments) share most of their elements and differ only in the head and tail. This phenomenon conforms readily with the characteristics of queue structure. We can redraw the segment space as in Figure 3 to depict this new relationship. Features (aka attributes) are to be attached to the queue structure. The idea is to let the queue structure traverse the search space and update its feature values accordingly. When the feature values satisfy the requirement, the content of the queue is an answer to the query. We name the underlying data structure an **attributed queue**.

Attributed queues not only conceptually capture the searching mechanism well, they also have an advantage in computational efficiency. Many features are **incrementally computable** in constant time. That is, the feature value of a segment is computable from its 'neighbor' and the difference by a function of lower complexity comparing to direct computation. On series of length n, for example, *length*, *sum* and *average* are $O(n)$ computable and incrementally constant-time computable. The Fourier transformation is $O(n \log(n))$ computable and incrementally $O(n)$ computable.

In reality, the size of a queue is always limited because of the time and space constraint of computers. Hence we name the model as **Limited Patience Patterns**.

The features of attributed queues are usually, but not necessarily, incrementally computable aggregations (computed from the old feature value and the data entry to be inserted or deleted without accessing the content of the queue). Incremental computation make the conceptual modeling more elegant. However, if the model only provides one queue as the buffer or only one pattern to be searched, the expressive power would be very poor because then users cannot define some basic components and use them to build more sophisticated patterns. So the LPP model provides some constructs to server the purpose of building **complex patterns**.

- **Merge of multiple time series**: Many time series can be merged to a single time series. By default, the entries of the same index are combined. If there is an ordered and unique field in each time series of same type, e.g. time or date, the merge can be done by aligning the entries according to the ordered field.

- **Matching multiple patterns over a single queue**: Users can specify multiple patterns to be searched at the same time; that is, an answer segment must be an instance of all the specified patterns simultaneously.

- **Serial composition of patterns**: A **composite pattern** can be defined as the concatenation of many simple patterns which are formed by single queues. That is to say, queues can be concatenated to form a more complex queue.

For the time being, we do not consider other constructs. It is our belief that most useful patterns can be constructed in LPP model by combining these three constructs.

Attributed queues can perform both approximate and exact retrieval. The degree of match, or 'metric' between patterns and segments, can be represented as just another attribute of the queue. Often this metric can be naturally expressed as a function of other features (attributes) of a segment. Formally, given a set of features of size N, we can define an N-dimensional topological space where each feature is a dimension. The metric is then just a suitable function of these features — whether a Euclidean distance measure or something more general.

The output of the attributed queues is a time series formed by the entries of attributes of the satisfied segments which, again, can be searched against or merged with other time series. Users must be aware that the output time series does not necessarily have an ordered (temporal) field even though the input time series does, because the ordered temporal attribute that users are interested in might not be the temporal field of the beginning element of the segments, and there is no ordering on all the segments that can preserve the original temporal ordering on temporal field of every segment.

Limited Patience Patterns and SQL/LPP differ markedly from previous work. Limited Patience Patterns use attributed queues which are variable-sized, hence better able to model patterns *than stream-based or transducer-based models. Queue-based models emphasize incremental, efficient evaluation, and seem well-adapted to practical problems.*

Although its design principles are grounded in the operational model of attributed queues, SQL/LPP can be developed as a purely declarative query language, with significant expressive power. SQL/LPP can express moving-window aggregates as well as subtle data patterns. SQL/LPP is extensible. Users can define new functions according to the principle of attributed queues. Furthermore, if users specify whether the user-defined functions have the properties we defined in Section 4.1, the user-defined functions can be included in SQL/LPP query analysis so that there is no "blind spot" in SQL/LPP query optimization.

3 SQL/LPP: The Language

The syntax of SQL/LPP follows the style of SQL, so SQL users will have short learning curves in adapting elements of the language to their work.

3.1 Extensional Data Model And Data Definition Language

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

Example 2 *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(
  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.

3.2 Pattern Functions

Pattern functions are the basic building blocks of SQL/LPP pattern specification. It is our intention to make the function set expandable. The functions we select to introduce in this section are basic but sufficient to define many interesting patterns.

For each segment s of type $seg(\mathcal{T})$ where $\mathcal{T} = \{c_1 : \tau_1, \cdots, c_n : \tau_n\}$, $C = \{c_1, \cdots, c_n\}$, and $|s| = \ell$, we define the following pattern functions regarding to segments:

1. $first : seg(\mathcal{T}) \times int \rightarrow \mathcal{T}$, $first(s,k) = e$ iff $index_s(e) = k$.

2. $last : seg(\mathcal{T}) \times int \rightarrow \mathcal{T}$, $last(s,k) = e$ iff $1 \leq \ell - k + 1 \leq \ell$ and $index_s(e) = \ell - k + 1$.

3. $count : seg(\mathcal{T}) \rightarrow int$, $count(s) = \ell$.

4. $sum : seg(\mathcal{T}) \times C \rightarrow \tau_i$, $sum(s, c_i) = r$ iff c_i is of type τ_i, the '+' (plus) operator is defined in τ_i, and $r = \sum_{e \in s} e.c_i$.

5. $avg : seg(\mathcal{T}) \times C \rightarrow \tau_i$, $avg(s, c_i) = t$ iff c_i is of type τ_i, the '+' (plus) and '/' (divide) operators are defined in τ_i, and $t = \sum_{e \in s} \frac{e.c_i}{\ell}$.

6. $max : seg(\mathcal{T}) \times C \rightarrow \tau_i$, $max(s, c_i) = u$ iff c_i is of type τ_i, the comparison operator '$\leq$' is defined in τ_i, there exists an element e in s such that $e.c_i = u$ and for all e in s, $e.c_i \leq u$.

7. $min : seg(\mathcal{T}) \times C \rightarrow \tau_i$, $min(s, c_i) = v$ iff c_i is of type τ_i, the comparison operator '$\leq$' is defined in τ_i, there exists an element e in s such that $e.c_i = v$ and for all e in s, $v \leq e.c_i$.

8. $sorta : seg(\mathcal{T}) \times C \rightarrow seg(\mathcal{T})$, $sort(s_1, c) = s_2$ iff s_2 is a permutation of s_1, sorted in ascending order according to the value of column c.

9. $sortd : seg(\mathcal{T}) \times C \rightarrow seg(\mathcal{T})$, $sort(s_1, c) = s_2$ iff s_2 is a permutation of s_1, sorted in descending according to the value of column c.

Because of frequent use, we define $first(s)$ and $last(s)$ as syntactic sugar for $first(s, 1)$ and $last(s, 1)$ respectively.

Given a segment s of type $seg(\mathcal{T})$ and an element e of type $\mathcal{T}$, we define the following pattern functions to deal with element reference.

1. $is\_first : \mathcal{T} \times int \rightarrow boolean$, $is\_first(e,k) = T$ iff $1 \leq k \leq \ell$ and $index_s(e) = k$

2. $is\_last : \mathcal{T} \times int \rightarrow boolean$, $is\_last(e,k) = T$ iff $index_s(e) = \ell - k + 1$.

3. $next : \mathcal{T} \times int \rightarrow \mathcal{T}$, $next(e_1, k) = e_2$ iff $index_s(e_1) = m$, $m + k \leq \ell$ and $index_s(e_2) = m + k$.

4. $previous : \mathcal{T} \times int \rightarrow \mathcal{T}$, $previous(e_1, k) = e_2$ iff $index_s(e_1) = m$, $m - k \geq 1$ and $index_s(e_2) = m - k$.

Also, we define $is\_first(e)$, $is\_last(e)$, $next(e)$ and $previous(e)$ as the syntactic sugar for $is\_first(e,1)$, $is\_last(e,1)$, $next(e,1)$ and $previous(e,1)$ respectly. These functions are chosen to be supported by SQL/LPP because they are useful in most applications. However, users can always extend the language by adding new functions if they cannot express queries by these functions.

3.3 Pattern Expressions, Pattern Formulae And Pattern Sentence

Using the functions defined above and usual mathematical operators and functions, we can construct **pattern expressions**. We require pattern expressions to be type-safe. For example, assuming s is a segment of the `quote` type defined in Example 2 and e is a variable, the following expressions are legal pattern expressions: $is\_first(e)$, $last(3).price$, $avg(s, volume) * count(s)$ and $next(first(s,3),3).low - previous(last(s,2)).high$.

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 $(\mathit{NOT}\ \mathcal{P})$, $(\mathcal{P}\ \mathit{AND}\ \mathcal{Q})$, $(\mathcal{P}\ \mathit{OR}\ \mathcal{Q})$, $(\mathcal{P}\ \mathit{THEN}\ \mathcal{Q})$, $[\mathit{ALL}\ e\ \mathit{IN}\ s](\mathcal{P})$ and $[\mathit{SOME}\ e\ \mathit{IN}\ s](\mathcal{P})$ 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 $ts[x,y] \models (\mathit{NOT}\ previous(first(s)).price > 0)$.

3.4 Searching 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 the pattern we described in Example 1. 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

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	$\boxed{\textbf{T}}$	T	$\boxed{\textbf{T}}$	F	$\boxed{\textbf{T}}$	F

Figure 4: 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 $first(s).price < last(s).price$.

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 3 *Consider a pattern* **profitable_period** *described by pattern sentence* $first(s).price < last(s).price$. *As shown in Figure 4, 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 5.

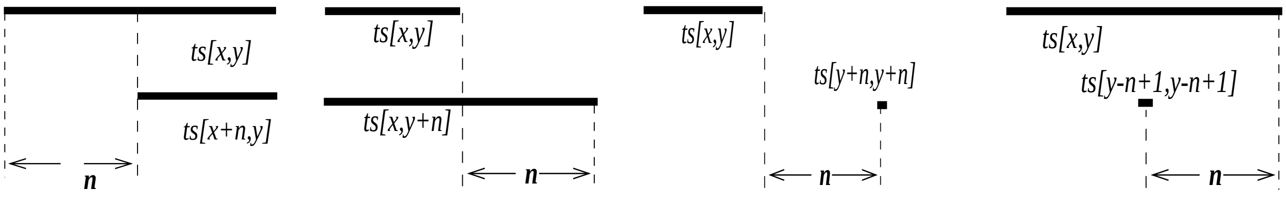
1. **SKIP START** n: $ts[x+n, max(x+n,y)]$.

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**: $succ(s)$.



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

Figure 5: The relative position between the most recent reported segment and the search restarting point under different restart search directives

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 in Example 1. 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.

3.5 Defining SQL/LPP 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.

3.5.1 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 4 *The pattern uptrend described in Example 1 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 5 *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
```

Expressing the 'median' is very difficult in most stream-based or transducer-based languages[13][11]. The next example shows SQL/LPP provides a comparatively simple way to express the idea.

Example 6 *The following definition defines a period with the final date and the median of closing price as the attributes.*

```
CREATE PATTERN median_seg(days int) AS
  SEGMENT s OF quote
  ATTRIBUTE date IS last(s).date
  ATTRIBUTE median IS first(sorta(s,close),days/2).price
  WHERE count(s)=days
```

This pattern[2] maintains a sorted segment. The median is the middle value in the sorted segment.

A minor modification of pattern *median_seg* can be used to compute any quantile[13]. Readers should bear in mind that median or quantile computation is expensive by its nature. SQL/LPP provides a way to express the concept, not a faster algorithm to compute it.

3.5.2 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:

[2] We assume division on the integers returns integers too, so that $2n/2 = n$ and $(2n + 1)/2 = n + 1$.

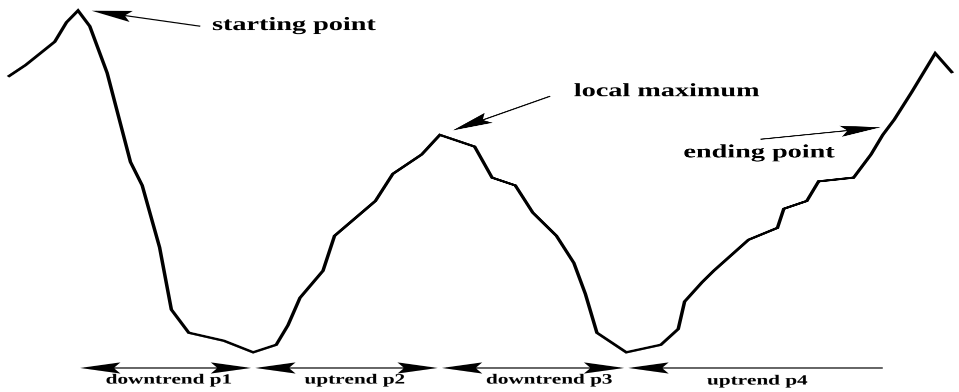


Figure 6: Double-bottom pattern

```
{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 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 7 *Assume pattern downtrend is defined symmetrically to pattern uptrend in Example 4. The pattern double_bottom consists 4 trends as shown in Figure 6. 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.

3.6 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 8 *Using pattern moving_avg_seg defined in Example 5, 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 8, 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 9 *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
```

3.7 Non-Time-Series Queries

Although SQL/LPP is designed for querying segments instead of individual records, users might want to issue non-time-series queries such as "Find all stocks priced above 100 on Feb. 5, 1998". Since the extensional database model is not ordinary. SQL queries cannot handle it well. SQL/LPP provides a mechanism to deal with the issue.

For every time series element type, SQL/LPP provides a system-defined pattern called *element*. It is a trivial pattern of length one and every column is an attribute of the pattern. For example, SQL/LPP define the following pattern for the row type in Example 2.

```
CREATE PATTERN element AS
  SEGMENT s OF quote
  ATTRIBUTE date   IS first(s).date
  ATTRIBUTE price  IS first(s).price
  ATTRIBUTE volume IS first(s).volume
  WHERE count(s)=1
```

The following example shows how to use the *element* pattern to perform a non-time series pattern query.

Example 10 *Find all stocks priced above 100 during 1996 and 1997. List the stock symbols, dates and closing prices.*

```
SELECT m.symbol,m.date,m.price
FROM daily_stock ds
BY SEARCHING element m in quotes
  START WHEN quotes.date >= DATE '1996-01-01'
  STOP  WHEN quotes.date >  DATE '1997-12-31'
WHERE  m.price >= 100
```

4 Query Optimization

SQL/LPP is significant in that it permits many important optimizations. For example, by observing the pattern `uptrend`, it is not hard to see that it is possible to skip some segments. A specially designed program can achieve time $O(n)$ complexity if `NON-OVERLAPPING` is specified. However, a mechnical optimization technique that works for generalized pattern query is not very intuitive. In this section we sketch some of the many results one can derive in static analysis (pattern-length and property analysis) and dynamic analysis of pattern formulae. These techniques are purely mechnical and based on syntatic analysis of query sentences.

4.1 Static Analysis of SQL/LPP Queries

Unlike most general purpose programming languages, SQL/LPP is constructed so that pattern sentences can be analyzed for query optimization. Optimization of SQL/LPP query evaluation (or pattern searching) has two goals. First, the search space should be reduced as much as possible. In some cases, if a node fails to satisfy the pattern sentence, it can be concluded that a set of other nodes will also fail, and hence there is no need to evaluate them. Second, redundant computation should be eliminated as much as possible. For some pattern sentences, after evaluating them on a node, it is possible to obtain the evaluation result on its neighboring nodes with little or no computation.

4.1.1 Pattern Length Analysis

As shown in Figure 2, it can be concluded that the most effective way to reduce the search space is to set narrow length limits.

Definition 4 *The* **lower-** *and* **higher length limit** *functions $\mathcal{L}$ and $\mathcal{U}$ map pattern expressions and formulae to integers representing the lower and upper length limits. Assuming s is a segment variable, e is an element variable, c is a column name, $\mathcal{E}_1$ and $\mathcal{E}_2$ are pattern expressions, $\mathcal{P}$ and $\mathcal{Q}$ are pattern formulae, n is an integer and limit is the system resource bound, then $\mathcal{L}$ and $\mathcal{U}$ are recursively defined on pattern expressions by:*

$$
\begin{array}{ll}
\mathcal{L}(e) = 1 & \mathcal{U}(e) = limit \\
\mathcal{L}(s) = 1 & \mathcal{U}(s) = limit \\
\mathcal{L}(count(s) > n) = n+1 & \mathcal{U}(count(s) > n) = limit \\
\mathcal{L}(count(s) \leq n) = 1 & \mathcal{U}(count(s) \leq n) = n \\
\mathcal{L}(count(s) < n) = 1 & \mathcal{U}(count(s) < n) = n-1 \\
\mathcal{L}(count(s) = n) = n & \mathcal{U}(count(s) = n) = n \\
\mathcal{L}(count(s) \geq n) = n+1 & \mathcal{U}(count(s) \geq n) = limit \\
\mathcal{L}(\mathcal{E}_1.c) = \mathcal{L}(\mathcal{E}_1) & \mathcal{U}(\mathcal{E}_1.c) = \mathcal{U}(\mathcal{E}_1) \\
\mathcal{L}(next(\mathcal{E}_1, n)) = \mathcal{L}(\mathcal{E}) + n & \mathcal{U}(next(\mathcal{E}_1, n)) = \mathcal{U}(\mathcal{E}_1) \\
\mathcal{L}(previous(\mathcal{E}_1, n)) = \mathcal{L}(\mathcal{E}_1) + n & \mathcal{U}(previous(\mathcal{E}_1, n)) = \mathcal{U}(\mathcal{E}_1) \\
\mathcal{L}(first(\mathcal{E}_1, n)) = n & \mathcal{U}(first(\mathcal{E}_1, n)) = \mathcal{U}(\mathcal{E}_1) \\
\mathcal{L}(is\_first(\mathcal{E}_1, n)) = n & \mathcal{U}(is\_first(\mathcal{E}_1, n)) = \mathcal{U}(\mathcal{E}_1) \\
\mathcal{L}(last(\mathcal{E}_1, n)) = n & \mathcal{U}(last(\mathcal{E}_1, n)) = \mathcal{U}(\mathcal{E}_1) \\
\mathcal{L}(is\_last(\mathcal{E}_1, n)) = n & \mathcal{U}(is\_last(\mathcal{E}_1, n)) = \mathcal{U}(\mathcal{E}_1).
\end{array}
$$

If an expression is not in any of the above forms, its lower limit is the maximum of the lower limits of its sub-expressions, while its upper limit is the minimum of the upper limits of its sub-expressions.

$\mathcal{L}$ and $\mathcal{U}$ are also recursively defined on pattern formulae by:

$$
\begin{array}{ll}
\mathcal{L}(NOT\ \mathcal{P}) = 1 & \mathcal{U}(NOT\ \mathcal{P}) = limit \\
\mathcal{L}(\mathcal{P}\ AND\ \mathcal{Q}) = max(\mathcal{L}(\mathcal{P}), \mathcal{L}(\mathcal{Q})) & \mathcal{U}(\mathcal{P}\ AND\ \mathcal{Q}) = min(\mathcal{U}(\mathcal{P}), \mathcal{U}(\mathcal{Q})) \\
\mathcal{L}(\mathcal{P}\ OR\ \mathcal{Q}) = min(\mathcal{L}(\mathcal{P}), \mathcal{L}(\mathcal{Q})) & \mathcal{U}(\mathcal{P}\ OR\ \mathcal{Q}) = max(\mathcal{U}(\mathcal{P}), \mathcal{U}(\mathcal{Q})) \\
\mathcal{L}(\mathcal{P}\ THEN\ \mathcal{Q}) = \mathcal{L}((NOT\ \mathcal{P})\ OR\ \mathcal{Q})) & \mathcal{U}(\mathcal{P}\ THEN\ \mathcal{Q}) = \mathcal{U}((NOT\ \mathcal{P})\ OR\ \mathcal{Q})) \\
\mathcal{L}([ALL\ e\ IN\ s]\mathcal{P}) = \mathcal{L}(\mathcal{P}) & \mathcal{U}([ALL\ e\ IN\ s]\mathcal{P}) = \mathcal{U}(\mathcal{P}) \\
\mathcal{L}([SOME\ e\ IN\ s]\mathcal{P}) = \mathcal{L}(\mathcal{P}) & \mathcal{U}([SOME\ e\ IN\ s]\mathcal{P}) = \mathcal{U}(\mathcal{P})
\end{array}
$$

Proposition 1 *Assume all pattern expressions are simple, i.e. contain no reducible expression such as $previous(next(e))$ or $next(next(e, 3), 2)$. Then $\mathcal{L}$ and $\mathcal{U}$ accurately compute the pattern length bound for a pattern*[3]*.*

[3] It is not too difficult to compute the length bound of reducible expressions, but because of the space limit of this paper, we avoid that discussion here.

4.1.2 Property Analysis

Definition 5 *Assume $\mathcal{P}$ is an atomic formula. $\mathcal{P}$ is* **invariable** *if when it evaluates to false on a node, it will also evaluate to false on a set of neighboring nodes. $\mathcal{P}$ is* **sustainable** *is when it evaluates to true on a node, it will also evaluate to true on a set of neighboring nodes. Each property can be further classified according to two properties:*

1. *$\mathcal{P}$ is* **d-invariable** *if $ts[x, y] \not\models \mathcal{P} \Longrightarrow ts[x+1, y] \not\models \mathcal{P}$, where $x < y$. If a d-invariable sentence evaluates to false on a segment, it must be also evaluate to false on those shorter segments with the same ending element. For example,* `last(s).price>100` *is d-invariable.*

2. *$\mathcal{P}$ is* **i-invariable** *if $ts[x, y] \not\models \mathcal{P} \Longrightarrow ts[x, y+1] \not\models \mathcal{P}$, where $x < y$. If an i-invariable sentence evaluates to false on a segment, it must also evaluate to false on longer segments with the same starting element. For example,* `first(s).price >100` *is i-invariable.*

3. *$\mathcal{P}$ is* **d-sustainable** *if $ts[x, y] \models \mathcal{P} \Longrightarrow ts[x+1, y] \models \mathcal{P}$ where $x \leq y$. If a d-sustainable sentence evaluates to true on a segment, it must also evaluate to true on shorter segments with the same ending element. For example,* `last(s).price >100` *is d-sustainable.*

4. *$\mathcal{P}$ is* **i-sustainable** *if $ts[x, y] \models \mathcal{P} \Longrightarrow ts[x, y+1] \models \mathcal{P}$, where $x \leq y$.If a i-sustainable sentence evaluates to true on a segment, it must also evaluate to true on longer segments with the same starting element. For example,* `first(s).price >100` *is i-sustainable.*

It is obvious that invariability and sustainability can be very useful in reducing the search space. So the next issue is to check whether a given sentence has these property. Following is the way to identify the properties by examining the syntax of the pattern sentences.

Proposition 2 *Assume $\mathcal{E}$ is an expression and $\mathcal{P}$ and $\mathcal{Q}$ are formulae. By extending the definition to expressions and formulae*[4]*, the properties can be decided by following recursive rules:*

1. *Variables and constants are d-invariable, i-invariable, d-sustainable and i-sustainable.*

2. *If $\mathcal{E}$ is d-invariable (resp. d-sustainable), then $last(\mathcal{E})$, $is\_last(\mathcal{E})$ and $next(\mathcal{E}, k)$ are d-invariable (resp. d-sustainable).*

3. *If $\mathcal{E}$ is i-invariable (resp. i-sustainable), then $first(\mathcal{E})$, $is\_first(\mathcal{E})$ and $previous(\mathcal{E}, k)$ are i-invariable (resp. i-sustainable).*

4. *The properties(d-invariable, i-invariable, d-sustainable and i-sustainable) are closed under mathematical functions and operators.*

[4] These properties generally do not have a simple meaning for expressions. We only apply the properties to expressions for the purpose of recursively deciding whether a formula has the properties.

5. *If $\mathcal{P}$ and $\mathcal{Q}$ are d-invariable (resp. i-invariable), and $\mathcal{R}$ is an arbitrary formula, then ($\mathcal{P}$ AND $\mathcal{R}$), ($\mathcal{P}$ OR $\mathcal{Q}$) and [SOME e IN s]$\mathcal{P}$ are d-invariable (resp. i-invariable).*

6. *If $\mathcal{P}$ and $\mathcal{Q}$ are d-sustainable, then so are ($\mathcal{P}$ AND $\mathcal{Q}$), ($\mathcal{P}$ OR $\mathcal{Q}$) and [ALL e IN s]$\mathcal{P}$.*

7. *If $\mathcal{P}$ and $\mathcal{Q}$ are i-sustainable, then so are ($\mathcal{P}$ AND $\mathcal{Q}$), ($\mathcal{P}$ OR $\mathcal{Q}$) and [SOME e IN s]$\mathcal{P}$.*

The use of these two properties will be discussed in the next subsection. Here we introduce another useful property called **uniqueness** which is also able to be classified to two types. Uniqueness is weaker than invariability and sustainability, but it is also useful in optimization.

Definition 6 1. *An expression $\mathcal{E}(e)$,where e is unbounded, is* **d-unique** *if for all x, y and i such that $x < y$, $x \leq i \leq y$ and $ts[x, y] \models \mathcal{E}(ts[i]) = k$ where k is a constant, then either $ts[x+1, y] \models \mathcal{E}(ts[i]) = k$ or $\mathcal{E}(ts[i])$ is evaluated to null in $ts[x+1, y]$.*

2. *An expression $\mathcal{E}(e)$,where e is unbounded, is* **i-unique** *if for all x, y and i such that $x \leq i \leq y$ and $ts[x, y] \models \mathcal{E}(ts[i]) = k$ where k is a constant, then $ts[x, y+1] \models \mathcal{E}(ts[i]) = k$.*

We extend the definition of uniqueness properties to formulae. It is comparatively easier to decide if an expression or a formula is unique.

Proposition 3 1. *An expression is* **d-unique** *if it does not contain any of the following functions: $first$, $count$, sum, avg, max, min, $sorta$, $sortd$ and is_first.*

2. *An expression is* **i-unique** *if it does not contain any of the following functions: $last$, $count$, sum, avg, max, min, $sorta$, $sortd$ and is_last.*

4.2 Dynamic Analysis of SQL/LPP Queries

With the result from length limit analysis and property analysis, a search engine can, in some cases, obtain the result of pattern sentence evaluation on a segment with very little or no computation. The following lemmas are the basis of SQL/LPP query optimization.

Lemma 1: If $\mathcal{P}$ is i-invariable and $ts[u, v] \not\models \mathcal{P}$, then $ts[u, y] \not\models \mathcal{P}$, for all $y \geq v$.

Lemma 2: If $\mathcal{P}$ is d-invariable and $ts[u, v] \not\models \mathcal{P}$, then $ts[x, v] \not\models \mathcal{P}$, for all $x \geq u$.

Lemma 3: If $\mathcal{P}$ is d-sustainable and $ts[u, v] \models$ [*ALL e IN s*]$\mathcal{P}$ then $ts[x, v] \models$ [*ALL e IN s*]$\mathcal{P}$ for all $x \geq u$.

Lemma 4: If $\mathcal{P}(e)$ is i-sustainable, $ts[u, v] \models$ [*ALL e IN s*]$\mathcal{P}(e)$ and $ts[u, v+1] \models (\mathcal{P}(s[v+1])$ then $ts[u, v+1] \models$ [*ALL e IN s*]$\mathcal{P}$.

Lemma 5: If $\mathcal{P}$ is d-unique and $ts[u, v] \models$ [*ALL e IN s*]($is\_first(e)$ *OR* $\mathcal{P}$) then $ts[x, v] \models$ [*ALL e IN s*]($is\_first(e)$ *OR* $\mathcal{P}$) for all $x \geq u$.

Lemma 6: If $\mathcal{P}(e)$ is i-unique, $ts[u,v] \models [all\ e\ in\ s](is\_last(e)\ OR\ \mathcal{P})$ and $ts[u,v+1] \models \mathcal{P}(s[v])$ then $ts[u,v+1] \models [ALL\ e\ IN\ s](is\_last(e)\ OR\ \mathcal{P}))$

Lemma 7: For all x, y, if $y - x + 1 < \mathcal{L}(\mathcal{P})$ or $y - x + 1 > \mathcal{U}(\mathcal{P})$ then $ts[x,y] \not\models \mathcal{P}$.

The above lemmas can be used in optimizing search procedure almost immediately. When applicable, lemma 1, 2 and 7 prune the part of the search space involving the pattern sentence, while lemma 3, 4, 5 and 6 allow search engines to move to neighboring segments without re-evaluating the pattern sentence against the whole segment. These lemmas can also be applied to sub-sentences when it is worthwhile. Here are examples applying the lemmas to query evaluation:

20-day Moving Averages: By length analysis and lemma 7, the search space can be pruned to $\{ts[1,20], ts[2,21], ts[3,22], \ldots\}$. Pattern searching is then linear-time computable. Further, search engines can find a path from $ts[x,y]$ to $ts[x+1,y+1]$ by one insertion and one deletion and the return value of avg can be updated quickly without adding 20 numbers. The computation is incremental, hence 2000-day moving averages can be evaluated as fast as 20-day moving averages. Without the LPP model, the time complexity can be $O(n^2L)$ where n is the length of the time series and L is the maximal length the system can handle. If $L > n$, then the time complexity is $O(n^3)$. In the LPP model, it is executable in time complexity of $O(n)$.

Uptrend: The subsentence $[ALL\ e\ IN\ s] is\_first(e)\ OR\ e.close \geq previous(e).close$ also require time complexity $O(n^2L)$ or (n^3) without the LPP model. However, by the optimization methods we proposed, it is easy to see the subsentence is i-invariable, d-unique and i-unique. So:

- Lemma 1 can be applied to prune all nodes with the same starting point and longer length than any one evaluating to **F**.
- Lemma 5 can be applied to assign **T** to all nodes with the same ending point and shorter length than any one evaluating to **T**.
- Lemma 6 can be applied to incrementally evaluate whether the sub-sentence remains **T** under the segments with the same starting point and longer length.

The complexity can be reduced to $O(n^2)$ which is optimal[5]. If search directive `NON-OVERLAPPING` is specified, the complexity is reduced further to $O(n)$.

Double Bottoms: Without the LPP model, a naive way to search this composite is to find every uptrend and downtrend, then join them together. The time complexity can be $O(4n^3 + n^{2^4}) = O(n^8)$. An immediate improvement is to sequentially find uptrend and downtrend to eliminate join operation so the time complexity can be reduced to $O(n^3)$. With the LPP model, the time complexity can be reduced to $O(n^2)$ or $O(n)$(if `NON-OVERLAPPING` is specified).

[5]Consider a time series of length n is itself a long uptrend, the each of its $\frac{n(n+1)}{2}$ segment is an uptrend.

We have studied optimization on more cases than these examples. In most cases, we can achieve optimal complexity. In some cases, the optimization technique can not be applied. For example, the pattern **Profitable Period** with pattern sentence $first(s).close < last(s).close$ represents the worst case. It does not have any of the properties we defined above and it does not have a user-defined upper length limit, hence no lemma is applicable to prune its search space. However, by closely examine this example, we can see the number of answers is $O(n^2)$ which is same as the time complexity SQL/LPP requires to evalute the query. Further, a SQL/LPP engine can find beforehand (right after it fails to find any length limit or property) that this query may require $O(n^2)$ time to evaluate and advise the users to add some length constraint to reduce the time complexity to $O(n)$.

5 Implementation

We have developed a prototype SQL/LPP system at UCLA. The basic design is shown in Figure 7. It is based on Java and JDBC in order to maximize cross-platform portability.

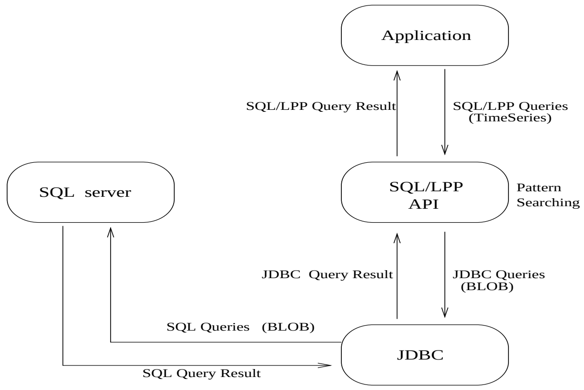


Figure 7: Design of SQL/LPP implementation

Time series elements are realized as objects, and time series are complex objects containing time series elements organized sequentially. The complex objects can be delivered as BLOBs (binary large objects) by utilize the persistence feature of Java objects. BLOB processing feature is present in every modern database system, so again the design is portable.

In the application programmers' view, the SQL/LPP engine is an API above JDBC. When this API receives SQL/LPP queries, it translates them to standard SQL queries, replacing time series retrieval by BLOB retrieval. The API then sends the SQL queries to a SQL server through JDBC. Upon receiving the query result, the SQL/LPP engine matches the patterns and returns the answer set to application programs. SQL/LPP is a superset of SQL, so those queries not related to time series are simply passed through the API.

Currently the *observer* software design pattern [8] is used extensively in processing SQL/LPP queries. Very briefly, attributed queues are observed by evaluation trees which compute the values of attributes, variables and pattern formulae. Evaluation trees are themselves observed by lemma entities. In turn, lemma entities provide performance-improving suggestions to a search commander, which reviews these suggestions and selects a traversal path in the search space.

In our current design, the SQL/LPP engine is a software layer above JDBC, so no indexing technique is applied. Given the optimization techniques we discussed in the previous section, the inheritantly larger searching space and the fairly strong expressive power of SQL/LPP, whether there is a feasible indexing method is still an open question.

6 Conclusion

Time series present new challenges for database systems. We discussed why existing query languages are unable to respond these challenges well. Although there are two very different approaches for specifying patterns — illustrative and descriptive — we have given a unified mathematical model that attains at least some of the advantages of both.

The main contribution of the research is the proposal of Limited Patience Patterns model and SQL/LPP language. LPP is an operational model based on attributed queues. The model fully utilizes the advantages of incremental computation and has favorable complexity than straightforward computation. SQL/LPP is a purely declarative query language which extends the standard query language SQL. The language provide some basic constructs for defining various patterns. Several patterns can be sequentially composed to give composite patterns.

We also described optimization techniques that reduce the computational complexity by analyzing the syntactic structure of queries and pruning the search space. An implementation plan of the system in a Java-based observer model was also discussed, demonstrating that the proposal is both practical and feasible.

References

[1] 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.

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

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

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

[5] 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.

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

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

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

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

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

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

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

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

[14] 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.

[15] C.S. Perng, D.S. Parker *SQL/LPP+: a Cascading Query Language for Temporal Correlation Verification in Time Series Databases*, Technical Report no.980036, UCLA, Computer Science, 1998.

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

[17] 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.

[18] 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.

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

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

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