A Sequential Pattern Query Language for Supporting Instant Data Mining for e-Services

Reza Sadri
Procom Technology
58 Discovery
Irvine, California 92618
sadri@procom.com

Carlo Zaniolo
Computer Science Dept.
University of California
Los Angeles, CA 90095
zaniolo@cs.ucla.edu

Amir Zarkesh **Jafar Adibi**
ZAIAS Technologies Corporation
12435 W. Jefferson Blvd. Suite 202
Los Angeles, CA 90066
azarkesh|jadibi@U4cast.com.

Abstract

Many e-commerce applications, including online auctions, personalization, and targeted advertising, require mining web-logs, transaction trails, and similar sequential patterns. Often, an "instant" response during an active trading session is required in critical applications. Therefore, e-services need efficient tools to perform fast, scalable searches for sequential patterns. Here, we describe SQL-TS, an extension of SQL that is highly optimized for searching patterns in sequences, and discuss its many uses in e-services.

1 Introduction

ZAIAS Technologies Corporation is a startup venture seeking to provide decision support and e-Services for web-based auctions. Toward this goal, ZAIAS is developing the technology to (i) monitor multiple ongoing auctions, (ii) determine the right price for auctioned goods, and (iii) secure well-priced items by placing timely bids. The determination of the right price for an item is based on historical and ongoing auction data, including those used to evaluate the expertise and bias of participating bidders. A recursive algorithm is then used to generate the Z-price as the best estimate of collective unbiased valuation of buyers [5]. Having determined the Z-price, the system can monitor ongoing auctions and place a winning bid for well-priced items matching the specific needs of the user, on a timely manner.

Time is critical, since premature bids can be counterproductive, and late bids can be in vain. Therefore, we have focused on technologies that can support instant data mining on streams of developing events, while searching databases for domain-specific knowledge and historical information using sophisticated queries. Time-series datablades currently supported in Object-Relational systems were viewed as too limited for expressing these sophisticated queries, and their procedural extensions were viewed as unsupportive of the fast development turnaround required by the application domain. Instead, we sought SQL extensions for complex time series queries, and query optimization techniques for their efficient execution. SQL-TS was found a valuable tool in online auction applications, and also other e-services discussed next.

A second application is mining web access logs. Our approach focuses on sessions, where a session is defined by a user coming to the site and quickly clicking through various pages (a session normally ends with a timeout after 10 or so minutes of inactivity). Typical applications include analyzing the effects of advertisements and promotions on users, and 'instant' profiling of a user into a cluster to which personalized contents and promotions can be targeted [1]. Fast response on a stream of developing events is also critical in security-oriented applications, in particular in detecting ongoing fraud or attacks by intruders. In the next section, we present SQL-TS through simple examples from the domains of online auctions an weblog mining. Then, we discuss security-related applications; in the last section, we briefly summarize the query optimization approach used in the implementation of SQL-TS.

2 SQL-TS

For instance, say that we have a stream of SQL tuples as follows (the tuples could come from a database table, or they could be processed on the fly as they are added to the weblog stream):

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

A user visiting the home page of the company starts a new session that consists of a sequence of pages

Proceedings of the 27th VLDB Conference, Roma, Italy, 2001

clicked; for each session number, SessNo, the log shows the sequence of pages visited—where a page is described by its timestamp, ClickTime, number, PageNo and type PageType (e.g., a content page, a product description page, or a page used to purchase the item).

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

Example 1 *Using the* FROM *clause to define patterns*

```
SELECT B.PageNo, C.ClickTime
FROM Sessions
     CLUSTER BY  SessNo
     SEQUENCE BY  ClickTime
     AS (A, B, C)
WHERE  A.PageType='content'
  AND  B.PageType='product'
  AND  C.PageType='purchase'
```

Thus, our Simple Query Language for Time Series (SQL-TS) is basically identical to SQL, but for the following additions to the FROM clause:

- A CLUSTER BY clause specifying that data for different sessions are processed as separate streams.
- A SEQUENCE BY clause specifying that the tuples for each SessNo are ordered by ascending ClickTime.
- The pattern AS (A, B, C) specifying that (for each session) we seek the sequence of the three tuples A,B, C (with no intervening tuple allowed) which must satisfy the conditions stated in the WHERE clause.

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

SQL-TS ability of recognizing patterns immediately as they occur, becomes important in auction monitoring. Assume that we have stream containing the data on ongoing bids as follows:

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

The objective is to purchase the product for a reasonable price. Then, we wait till the last 15 minutes before the closing, and we place an offer as soon as the stream of bids is converging toward a certain price. We detect convergence by a succession of three bids that raise the last bid by less than 2%.

```
SELECT T.auctn_id, T.time, T.amount
FROM bids
CLUSTER BY auctn_id
           SEQUENCE BY time
           AS (X,Y,Z,T)
WHERE   Y.amount - X.amount < .02 * X.amount
    AND Z.amount - Y.amount < .02 * Z.amount
    AND T.amount - Z.amount < .02 * Z.amount
```

The previous query is satisfied if the amount in Y is less than 2% above the amount in X, and also the same holds between Z and Y. To assure that we are within 15 minutes from closing, we must refer to a database table where the auctions are described:

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

Our query becomes:

```
SELECT T.auctn_id, T.time, T.amount
FROM  auction AS A,
      bids CLUSTER BY auctn_id
      SEQUENCE BY time
      AS (X,Y,Z,T)
WHERE   A.auctn_id = X.auctn_id
    AND X.time + 15 Minute > A.deadline
    AND Y.amount - X.amount < .02 * X.amount
    AND Z.amount - Y.amount < .02 * Z.amount
    AND T.amount - Z.amount < .02 * Z.amount
```

SQL-TS also supports the definition of repeating patterns using the star construct. For instance, to determine the number of pages the user has visited before clicking a 'product' page we simply write:

```
SELECT  count(*A)
FROM Sessions
     CLUSTER BY  SessNo
     SEQUENCE BY  ClickTime
     AS (*A, B)
WHERE A.PageType <> 'product'
  AND B.PageType = 'product'
```

Thus, *A identifies a maximal sequence of clicks to pages other than 'product' pages. Then, count(*A) counts those pages and returns them to the user. In addition to traditional SQL aggregates producing final returns, SQL-TS also supports rollups, running aggregates, moving-window aggregates, online aggregates, and other advanced aggregates used for time series analysis [1]. For instance, the following query identifies sessions where users have accumulated too many clicks, or spent too much time, without purchasing anything.

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

Here, ccount denotes a continuous count that is therefore increased by one for each new tuple in the sequence *X. Also, first(X) is a builtin aggregate that returns the first tuple in the sequence *X.

Thus, the recognition of *X continues while (i) there is no purchase, (ii) the length of *X consists of less than 100 clicks, and (iii) the time elapsed is less than 20 minutes. Once any of these conditions fails the sequence *X terminates. At the next click (assuming that this is not a 'purchase' page) SessNo is returned. (Triggering a time-out message to the remote user, who is asked to login again to continue.) Thus the final aggregates, such as count(*X), are applied at the end of the *X sequence and play no role in defining the length of the sequence. The continuous count ccount(X) (no star in the argument!) is instead computed during the sequence and can part take in the definition of the sequence itself.

3 Detection of Fraud and Intrusion

Stolen credit cards, theft of cellular phones and user ids, and similar crime cost e-services billions of dollars. Thus, instant, reliable detection of developing fraud would be very desirable, although very difficult to achieve. The three main problems in fraud detection are Evidence Extraction, Link Discovery and Pattern Discovery. Here, we only consider the third problem that often relies on outlier detection. For instance, typical patterns in credit card fraud relate to unusual customer shopping behavior such as *buying 2-3 TV set in one day*. Scalability is required with respect to the size and complexity of the pattern and to the amount of data in the stream—along with frequent access to the database containing customer information, product and historical data. Thus the requirements are similar to those of application domains previously discussed.

A simplified example might consists of a credit card company monitoring the customers' transactions. The total daily charges accumulated by a customer are represented using the following schema:

```
    log (Date, AccountNo, Amount)
```

Here, AccountNo is the account number, and Amount is the amount charged to the account at the specified Date. Then, to track the average spending during a specified period and detect when the spending increases considerably for 2 consecutive days, we can write:

```
SELECT Z.AccountNo, Z.Date
FROM   spending  CLUSTER BY AccountNo
                 SEQUENCE BY Date
                 AS (*X, Y, Z)
WHERE  COUNT(*X)=30
  AND  Y.Amount > 5 *  AVG(*X.Amount)
  AND  Z.Amount > 5 *  AVG(*X.Amount)
```

Prompt detection and response is also critical in security applications. Consider for instance a large enterprise, where many workstations are distributed at various locations, from which a central databases can be accessed via the company intranet and web browsers. The central database information is sensitive, and access restrictions are enforced via the usual login and password mechanisms. Security is a major concern, and the organization is seeking effective means to detect and deter attacks and intrusions in real time. In particular, repeated login failures occurring within a short time interval indicate a possible attack in progress. Detecting a fast series of unsuccessful login is simple, when these are issued from the same workstation. However, a smarter intruder, rather than trying many times from the same workstation, will walk to the workstation in the next room and try a few logins from there—then move to a workstation nearby and repeat the attempt from there, and so on. The question is how to detect such a peripatetic intruder. Since real time information about logins is available, the challenge is how to detect an ongoing attack by recognizing a particular spatio-temporal pattern of failed logins.

The log can be viewed as a temporally ordered relation, with the following schema:

```
    log (Etime, Etype, Build, Loc)
```

Here, Etime is a timestamp identifying the date and time of the event; Etype is the two-byte code for the event type (e.g., ls, lf, and to, respectively denote login success, login failure, and timeout events). Also, Build is the building code, and Loc is an integer encoding the xyz location within a building.

An attack is defined as the succession of three login failures occurring in physical and temporal proximity in the same building. (Since buildings have separate security and locations in different buildings fail the 'isnear' test, we can group by building and restrict our search to events that occur within buildings.) Then, we have the following SQL-TS query:

```
SELECT Z.Build, Z.Loc, Z.Etime
FROM  log   CLUSTER BY Building
            SEQUENCE BY Etime
            AS (*X, *Y, Z)
WHERE  ( (X.Etype='lf' AND ccount(X)=1)
         OR X.Etype <> 'lf'
         OR near(X.Loc, first(X).Loc)=0)
   AND ((Y.Etype='lf' AND ccount(Y)=1 AND
        first(X).Etime > first(Y).Etime-7 Minute)
       OR Y.Etype <> 'lf'
       OR near(Y.Loc, first(Y).Loc)=0 )
   AND  Z.Etype='lf' AND
       Z.Etime - 7 Minute < first(Y).Etime
```

Observe that the first event in *X, identified by the condition ccount(X)=1, must be an 'lf' event. Then, the rest of *X simply moves trough events that, either

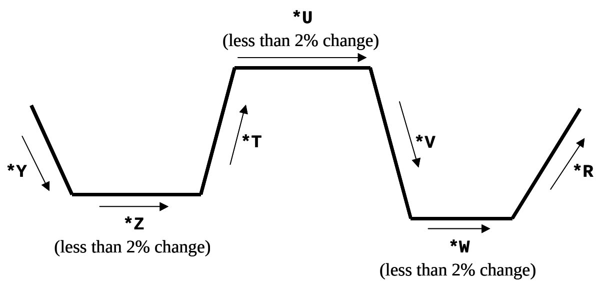

Figure 1: **The relaxed double bottom pattern.**

because they are not of type `'lf'` or occur at workstations not near that of this first event, are not related to the initial login failure. Then, `*Y` detects a second login failure, near that of `first(X)` and less than seven minutes after that. Finally, `Z` finds that a third `'lf'` event has occurred at a nearby workstation and within seven minutes of the last one. At this point, a likely attack by a peripatetic intruder is detected and the location of this last workstation is returned to trigger a security alarm.

4 Efficient Implementation

To achieve fast response on instant data mining queries, and scalability on large data sets, SQL-TS relies on an efficient implementation based on a novel query optimization technique called OPS [3]. OPS can be viewed as a generalization of the algorithm proposed by Knuth, Morris and Pratt (KMP) to optimize search [2] for strings. For instance, the query of Example 1 specifies a search for the sequence of values `'content, product, purchase'` in three successive records `A,B,C`. Now, if `'content'` and `'product'` are found in first two records, but `'purchase'` is not found in the third one, a naive algorithm would backtrack to the second record looking for `'content'`. But this cannot succeed given that, in the previous scan, the second record was found to contain `'product'`; thus, the KMP algorithm instead restarts from the third record. In general, by avoiding unnecessary backtracking, KMP assures that the cost of finding a pattern in a given text is linear in the sum of the sizes of the pattern and text, rather than their product [2]. The OPS algorithm [3] generalizes this idea to work with arbitrary SQL-TS queries that can be much more complex than those treated by KMP, since SQL-TS queries can also contain (i) `CLUSTER BY`, (ii) general predicates (such as `Z.price < 0.80*Y.price`), (iii) boolean expressions of such predicates, (iv) repeating patterns expressed by the star, (v) aggregates on such repeating patterns. OPS deals effectively with these added complexities, often with dramatic improvements. For instance, we expressed in SQL-TS the relaxed double bottom pattern [1] of Figure 1, and used it to search the Dow Jones Industrial Averages for the last 25 years. As shown in Figure 2, twelve

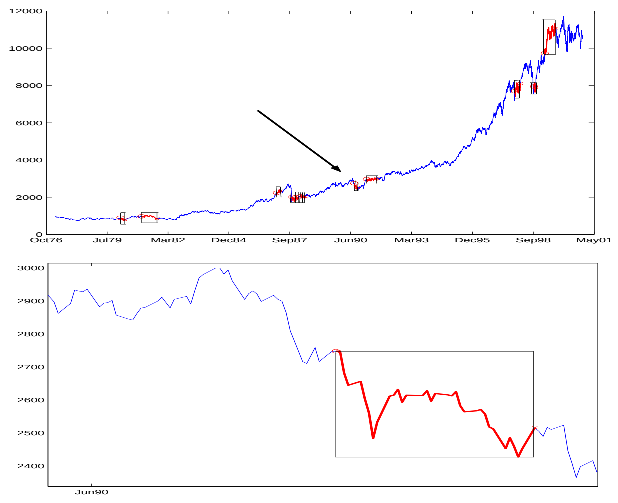

Figure 2: **Doublebottoms found in the DJIA data are shown by boxes. The bottom picture is zoomed for the area pointed by arrow in the top picture and shows one of the matches.**

matches were found. The figure also shows the double bottom that occurred around June 1990. The OPS optimization for the relaxed double bottom query, yields a search 93 times faster than naive search. For some other queries with complex search patterns, we obtained 800-fold speedups. The ongoing SQL-TS implementation builds upon the AXL system, which is also used to define the many new aggregates types required by time series [4].

Acknowledgements. The authors are grateful to Dick Tsur for his encouragement and illuminating discussions that led to the writing of this paper. This work was partially supported by the National Science Foundation under grant: NSF–IIS 0070135.

References

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

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

[3] Reza Sadri et al., Optimization of Sequence Queries in Database Systems. PODS 2001.

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

[5] ZAIAS Technologies Corporation. Recursive Z-prices over related auctions transaction graph, White paper 03–01–16-2001, Jan. 2001.