Event-Oriented Data Models and Temporal Queries in Transaction-Time Databases

Carlo Zaniolo
University of California, Los Angeles
zaniolo@cs.ucla.edu

***Abstract*—Past research on temporal databases has primarily focused on state-based representations and on relational query language extensions for such representations. This led to many different proposals that had did not succeed in making a significant impact on SQL-compliant DBMS. More recently however, there has been significant interest and progress on event sequences, leading to vendor-proposed extensions of SQL standards for pattern queries based on Kleene-closure expressions. In this paper, we first outline these extensions and their uses in dealing with sequence of events, and then show that they can also be used effectively to express more traditional temporal queries, such as coalescing and joins, on state-based representations. Thus, we propose an approach that takes full advantage of the fact that every state-based representation also has a dual representation based on its start-event and its end-event.**

I. Introduction

Most of past research on temporal databases has focused on state-based representations and query extensions on temporal data models based on these representations. In spite of their great depth, breadth [1], [2], [3] and technical elegance [4], [5], [6], these state-oriented approaches failed to make a real impact on commercial database standards. A discussion of the reasons (technical or otherwise) that prevented the successful adoption of the various state-oriented approaches is outside the scope of this paper, which instead focuses on the changed situation and the many opportunities that newly proposed SQL standards offer for event-oriented temporal reasoning.

In terms of SQL standards, a first interesting development is that several DBMS vendors are proposing new specs for transaction-time databases [7]. This development is significant, because a transaction time database represents the quintessential historical database designed to preserve a reality that was truly valid in the past (as opposed to the human recreated history of a valid-time database). Furthermore, for web documents, transaction time often plays a role similar to that played by valid time in traditional databases. For instance, in a web information system, such as Wikipedia, the valid time of a page corresponds to the time in which the page became available on the web, and this time corresponds to the start transaction time of the information contained in the underlying DBMS. In this paper, we focus primarily on transaction-time databases; however similar queries could also be used on valid-time and bi-temporal databases.

The most important development in terms of emerging SQL standards, is the new specs proposed for finding patterns in ordered sequences of events using Kleene-closure constructs [8], [9] finds important applications on both databases and data streams. In this paper, we show that the power and flexibility brought by these new constructs allow us to express complex temporal queries on stat oriented representations; toward that goal, we exploit the event/state-based duality that comes so natural in the real world where, e.g., a salary-increase event and the state of being at the new salary level are viewed as two facets of the same reality. For temporal databases, the event-oriented extensions of SQL proposed in [8] are significant at the practical level and the conceptual level. Indeed, obvious practical benefits follow from the fact that the new SQL standards now pushed by DBMS vendors will soon provide better support for temporal queries and reasoning in information systems. At the conceptual level, these developments force us to recognize that events can provide a more robust ontological basis for temporal databases than states. Indeed, many state-based representations have been championed by some researchers but discounted by others [10], including (i) maximal periods attached to each tuple, (ii) set of such periods attached to each tuple, (iii) attribute time-stamping, and (iv) point-based representations. Therefore, while there are many appealing state-oriented views, there is not a unique one. Event-based representations are instead immune from this proliferation of approaches: to the best of our knowledge, there is only one natural representation for an event— i.e., the one where the time stamp stating when the event occurred is added to the other information describing the event. Furthermore, this very representation of events is used in many related technical areas—e.g., active databases— whereas, the state-based representations are primarily artifacts of temporal databases. Thus, in this paper, we adopt the ontological realignment of (a) using events as our fundamental primitives, and (b) state-based representations as views of convenience to facilitate temporal queries. Indeed, we will show that

(i) some temporal queries are best expressed in SQL:1992 [11] using basic state-based representations,
(ii) other temporal queries, e.g., aggregate queries, can be handled by SQL:2003 OLAP functions on event sequences [12], [13], [14], and finally,
(iii) Many complex historical queries are best expressed using the Kleene-closure extensions of SQL [8] and unified views that combine features from (i) and (ii).

The paper is organized as follows. After briefly reviewing the related work in the next section, we proceed to discuss

how evolution has transformed SQL into a language that, though OLAP functions and Kleene-closure extensions, can express and support efficiently complex temporal queries on ordered sets and sequences. Thus, in Section IV, we discuss how different temporal queries are best supported through different views, and introduce a unified historical view that is particularly supportive of Kleene-closure queries. In the last section, we briefly discuss implementation issues and our current work on a transaction-time data base system.

II. Related Work

A number of practical applications has motivated the continuous interest in transaction-time databases [15], for which efficient indexing and implementation techniques have been proposed [16], [17], but face many of the query language complexities faced valid-time databases [10]. This is exemplified by existing systems such as Flashback [18] and ImmortalDB [16] that allow users to rollback to old versions of tables (e.g., to correct old errors), but do not provide complex temporal query support. Another problem shared by both transaction time and bitemporal databases is that of schema evolution, insofar that archival quality can only be achieved by storing historical data under the schema under which it was originally created [19].

Some recent research work has recently focused on the problem of representing transaction-time information in XML. In [20], valid time on the Web is supported by proposing a new <valid> markup tag for XML/HTML documents, thus temporal visualization can be implemented on web browsers with XSL. In [21], a dimension-based method is proposed to manage changes in XML documents, however how to support queries is not discussed. Other recent approaches aim at supporting temporal XML documents through extending XML data models or query languages, such as extending XML data model or XPath to support temporal XML documents in [22], [23] and [24]. A τXQuery language is proposed in [25] to extend XQuery for temporal support, which has to provide new constructs for the language.

In terms of temporal queries on events, the need to support pattern matching in sequences of events has motivated early work in query languages such as SEQ [26], [27], srql [28], and SQL/LPP [29]. A major step forward was then achieved with the introduction of Kleene-closure expressions in SQL-TS [30], [31], [32]. A growing application demand has recently brought DBMS vendors and DSMS startup companies to join forces and propose the PATTERN SQL for inclusion in the standards [8]. This interest reflects the recognition by industry that pattern queries have a wide range of applicability in areas such as financial services [33], RFID-based inventory management [31], click stream analysis [31], publish-subscribe [33], and electronic health systems [34]. Interest in pattern queries is also growing among researchers, who have proposed novel techniques for the optimization of such queries [35].

III. Temporal Evolution of SQL

The challenges faced by SQL-2 in expressing temporal queries have been elucidated by previous database researchers [1], [2]. But since then, SQL has undergone a significant evolution with the introduction of OLAP functions in SQL:2003 [12], [13], and even more significantly, the recently proposed new standards for event patterns [8]. Although these extensions were not motivated by temporal applications, they actually facilitate the expression of temporal queries to an extent that might surprise many database researchers along with the proposers of those extensions.

The main paradigm shift associated with SQL:2003 OLAP functions is the introduction of an explicit order to be exploited in the query search conditions—whereas ORDER BY statements in SQL-2 can only be used to re-order results produced by query conditions applied to unordered sets of tuples. An application of these new constructs pertains to the efficient computation of temporal aggregates such as count, sum, and coalesce functions on state oriented representations [36]. The solution approach described in [36] consists in (1) recasting each tuple timestamped with its validity period into a pair of the start-event and the end-event, (2) ordering the o the resulting tuples by their time stamps using a ORDER BY, (3) applying a window aggregates, where (5) the GROUP BY construct of the temporal aggregate is reexpressed using PARTITION BY.

An even more significant development is represented by the recently proposed new SQL constructs to identify patt in data streams and stored sequences using Kleene-closure expressions [8]. Kleene-closure extensions for SQL were first introduced by SQL-TS [32], and for simplicity of presentation, we will in fact use a more concise syntax based on that of SQL-TS to express our examples, including the following one that is based on the running example used in [8]:

Example 1: Let **Ticker(Symbol,Tstamp,Price)** represent historical stock prices: **Symbol** is a character column, **Tstamp** is a timestamp column (for simplicity shown as increasing integers) and **Price** is a numeric column. It is desired to partition the data by **Symbol**, sort it into increasing **Tstamp** order, and then detect the following pattern in **Price**:

- one or more consecutive falling prices,
- followed by a rise in price that goes higher than where the price was when the fall began.

Upon finding such patterns, it is desired to report the starting time, starting price, inflection time (last time during the decline phase) price, end time, and end price.

This complex query can then be expressed as follows:

```
SELECT A.Symbol, A.Tstamp, A.Price,
       max(B.Tstamp), min(B.Price),
  FROM Ticker AS PATTERN (A B+ E* F+)
  PARTITION BY Symbol ORDER BY Tstamp
  WHERE B.price < PREV(B.price) AND
    E.Price >= PREV(E.Price) AND E.Price < A.Price
  AND F.Price >= PREV(F.price) AND F.price >= A.price
```

This query uses the PARTITION BY and ORDER BY constructs of SQL:2003 OLAP functions [12], [13] to view the data as separate sequences, one for each **Symbol** value, all ordered by **Tstamp**. From these, **pattern** is used to define the search for patterns consisting of events that immediately follow each other—a condition that in standard SQL must be expressed very inefficiently as "there is nothing in between" condition. For instance, PATTERN**(X Y Z W)** would require three NOT EXISTS conditions (one between each pair of successive pattern variables) in addition to the conditions specifying the three joins on **Symbol**[1]. In terms of expressive power, a critical extension is represented by Kleene-closure conditions such as **B+** (denoting one or more occurrences of **B**) and **E*** (denoting zero or more occurrences of **E**), which would require the use of recursive queries if they had to be expressed in SQL:2003.

Powerful query optimization techniques are also available for these languages [32]. Among the exciting opportunities for new applications that have been created by these pattern languages particularly interesting one is trajectory identification.

Trajectories. Recent progress on satellite, sensor, RFID, video, and wireless devices has made it possible to systematically track object movements and collect huge amounts of trajectory data, e.g., animal movement data, vessel positioning data, and hurricane tracking data [37], [38], [39], [40]. As a result, trajectory data are now very common in the real world.

Trajectory classification, defined as the process of identifying moving objects based on their trajectories and other features find many applications in the real world [41], [42]. Accordingly, there is an ever-increasing interest in performing data analysis over trajectory data and identifying an object by the similarity of its trajectory to that of objects of known type. Past approaches on the retrieval of similar trajectories generally use distance functions that consider the distances between pairs of trajectories across time [37], [38], [39]. So, typical approaches use the shapes of whole trajectories, and match this shape with that of prototypes using distance functions and hidden Markov models to capture the evolution between states. More recently, efforts have been made to elevate the trajectory analysis, beyond the purely physical, and more toward the logical level. For instance, the approach proposed in [42] consists in (i) detecting discriminative segments (phases) in the these spatio-temporal trajectories and (ii) describing the whole trajectory as a well-defined sequence of various phases. Inasmuch as these discriminative features often find semantically significant interpretations, this approach bring us one step closer to analyzing and understanding trajectories in a way that humans do. For example, vessel detection and classification from satellite imaging sensors is or could be used

[1]An alternative way to express PATTERN**(X Y Z W)** could be to assign sequence numbers to the separate streams, via the **seq()** function, and require that the sequence number of Y, Z, and W, respectively, exceeds by one those of X, Y, and Z.

EmpHist:

empno	Salary	Title	deptno	start	end
1001	60000	Engineer	d01	1995-01-01	1995-06-01
1001	70000	Engineer	d01	1995-06-01	1995-10-01
1001	70000	Sr Engineer	d02	1995-10-01	1996-02-01
1001	70000	Tech Leader	d02	1996-02-01	1996-12-31

Fig. 1. A fragment of the history of the employee table.

for a number of applications: fishery control, pollution control, border control including illegal immigration and smuggling, maritime safety, search and rescue, piracy prevention, security of maritime trade routes, and anti-terrorism. As discussed in [42] vessel types can be classified on the basis of their different trajectories. For instance a simple trajectory of a fishing boat would be as follows:

Example 2: The basic trajectory of fishing boats:

(A) vessel leaves port P,
(B) vessel travels keeping a steady direction and speed,
(C) vessel arrives at the fishing area and crisscrosses the area for a while,
(D) vessel travels (back) keeping a steady direction and a good speed, and
(E) vessel returns to port–typically port P.

Kleene closure expressions can be used very effectively to characterize this kind of itinerary using a pattern such as **(A B+ C+ D+ E)**, where **A** denotes the boat being at the port, or near it, phases **B+** and **D+** are characterized by the boat travelling at about constant speed and direction, and phase **C+** is the boat crisscrossing the fishing area. Detailed constraints capturing the specific semantics of the situation at hand can be easily added as conditions on time, space, and velocity. Moreover this is a high level description of fishing trajectories that remains valid independent of the port the vessel leaves and the direction in travels toward—whereas previous approaches based on actual physical trajectories will fail to detect the similarity between vessels leaving from different ports, or vessel leaving from the same port but travelling to different fisheries. In fact, Kleene-closure entails logical characterization of spatio-temporal patterns that are general enough to be used in other applications: for instance, the fishing-boat trajectory when **C** is not a fishing area, could indicate suspicious activities, such as searching for sunken ships, or the placement of mines or sensors. Thus, different kinds of ships follow different trajectories [42]; moreover, different trajectories can be characterized by different Kleene-closure signatures. Therefore, the Kleene-closure extensions of SQL can be used in new and interesting time-oriented applications; moreover, as we shall see next they can be used to to support effectively more traditional temporal queries.

IV. Temporal Views and Temporal Queries

Consider now a more traditional-state based representation such that of Figure 1, below, that shows a short fragment of the evolution history of the salary-title-department table for employee **1001**.

In the representation of Figure 1, tuples are time-stamped with their maximal periods of validity; thus, a new tuple

is generated whenever our employee changes his/her salary, title, or department. This state-based representation provides a very useful view for expressing many temporal queries. For instance, a query such as *"What were the salary and title of employee 1001 on 1995-09-15,"* can be expressed via a simple selection. Also snapshot queries such as *"List all the department where employee 1001 ever worked,"* are easily expressed against this view. Also queries only requiring selection and temporal slicing (e.g., retrieve all the columns for the histories of certain employees over a given time period) can be expressed easily on this view. However, it is well-known that major problems are encountered in queries that require the elimination of columns by performing a projection operation. In fact, as a result of projections, some tuples become identical except for their periods of validity, which must therefore be coalesced. For instance in our example, Figure 1, the first two tuples are coalesced once salary is projected out. Therefore, the simple operation of projection on tuple-timestamped relations will now require the operation of coalescing—an operation whose intrinsic complexity is exacerbated by the limitations of SQL-2, where the operation can only be expressed via complicated statements that are prone to inefficient execution [43]. Therefore, eliminating or reducing the need for coalescing represents a key objective in many temporal database approaches.

A simple way to greatly reduce the need for coalescing is to decompose our employee relation according to their key obtaining the following three tables (that we will call H-tables), shown in Figure 2. Besides, reducing the need for coalescing, the H-tables representation can be used as the basis for efficient implementation [44]. On the other hand, this representation generates many tables, and therefore temporal joins on such tables must be used to answer typical queries involving several attributes. Although simpler than coalescing, expressing temporal joins in SQL is not without complications, since it involves the conditions that matching tuples have overlapping validity periods.

EH1:

empno	start	end	salary
1001	1995-01-01	1995-06-01	60000
1001	1995-06-01	1996-12-31	70000

EH2:

empno	start	end	Title
1001	1995-01-01	1995-10-01	Engineer
1001	1995-10-01	1996-02-01	Sr_Engineer
1001	1996-02-01	1996-12-31	Tech_ Leader

EH3:

empno	start	end	deptno
1001	1995-01-01	1995-10-01	d01
1001	1995-10-01	1996-12-31	d02

Fig. 2. The decomposition of the original table into three H-tables

In the topical literature, there has been much discussions on the pros and cons of these representations, and the many others that have been proposed. For all past discussions, however, little consideration has been given on using event-oriented query languages instead of state-oriented query languages. Indeed, while it is natural to view a state as the pair of start/end events, it was believed that this view would not be beneficial in overcoming the problems of SQL with complex temporal queries. The arrival of PATTERN SQL [8] is changing all that, since it facilitates the expression of both event-oriented queries and state-based queries on views such as that of Figure 3.

EmpUH:

empno	start	end	Salary	Title	deptno	Type
1001	1995-01-01	1995-06-01	60000	?	?	1
1001	1995-01-01	1995-10-01	?	Engineer		2
1001	1995-01-01	1995-10-01	?	?	d01	3
1001	1995-06-01	1996-12-31	70000			1
1001	1995-10-01	1996-02-01	?	Sr_Engineer	?	2
1001	1995-10-01	1996-12-31	?	?	d02	3
1001	1996-02-01	1996-12-31	?	Tech Leader	?	2

Fig. 3. Unified history table for employees

The **EmpUH** table of Figure 3 (where **UH** stands for 'unified history') simply represents the history of the atomic events that have occurred on our **Emp** table along with the time in which the state brought by this event ended. Once we add an additional column named **Type** to the H-tables in Figure 2, and fill them respectively with the values 1, 2, and 3, then see that **EmpUH** can be specified as the outer-join of these three tables (taking the equijoin on **empno** and **Type**, only).

Since no two tuples from different tables can share the same **Type** value, our outer join reduces the union of tuples from the H-tables padded with null values as needed (nulls are represented by "?"). The original H-tables can thus be derived back by the simple operations of projection and selection (to eliminate tuples with null values). Thus, for instance, our H-table **EH2** can be computed as follows:

Example 3: Columns Eliminated by Selection+Projection

```
SELECT empno, start, end, Title
FROM EmpUH
WHERE Type = 2
```

Thus the additional condition on **Type** is all that is needed when taking this projection, and as in the case of Figure 2, no coalescing is needed. However the view of Figure 3 is much more conducive than that of Figure 2 for complex queries involving evolution histories. This can be illustrated by the following example.

Example 4: Find employees who have risen quickly without changing department. More specifically, we want to find employees who

1) Once hired (with some salary and title and into some department),
2) have gone through one or more salary adjustments, followed by
3) a change in title followed by
4) a transfer to another department,
5) for a final salary that 40% above the initial one.

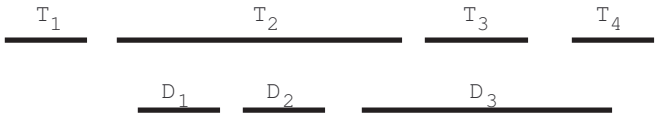

Fig. 4. Temporal Joins for T and D Histories

EmpTD

empno	Title	deptno	start	end
1001	Engineer	d01	1995-01-01	1995-10-01
1001	Sr Engineer	d02	1995-10-01	1996-02-01
1001	Tech Leader	d02	1996-02-01	1996-12-31

Fig. 5. The table employee-hist

For each such employee, show the **empno**, the department, the salary and timestamp at the start of the sequence, and those at the end of the sequence.

This complex query can be expressed in SQL-TS as shown in Example 5, where the PARTITION BY and ORDER BY reflect the arrangement of the data in Figure 3. Therefore, the history of each employee is processed as a separate sequence, where events are listed by their **start** time and then by **Type**. Thus, in the case of a new employee **Salary**, **Title** and **deptno** are listed in that order, i.e., they form the sequence **A B C** but share the same **start** time stamp (as per WHERE conditions). After the first three events connected with the hiring of the employee our pattern specifies **D+** that denotes one or more salary changes of condition (2). After that we have **E** and **F** that respectively correspond the conditions (3) and (4). Thus, the first six tuples in the table of Figure 3 would actually satisfy our pattern query, if were not for the final **Salary** condition.

Example 5: A Pattern of Salary-Title-Department History

```
SELECT A.empno, A.dept, A.Salary,
       A.start, last(E.Salary), E.start
FROM EmpUH AS PATTERN(A B C D+ E F)
PARTITION BY empno ORDER BY Tstamp, Type
WHERE A.Type= 1 AND B.Type =1 AND C.Type=3
AND A.start= B.start AND B.start =C.start
AND D.Type= 1 AND E.Type=2 AND F.Type=3
AND last(D.Salary)>1.4* A.Salary
```

Therefore, PATTERN SQL represents a very powerful event-oriented temporal language, i.e., the very domain for which it was designed. However its domain of effectiveness extends to state-oriented queries, starting with the temporal join and coalescing that have been the focus of so much travail and discussion in temporal databases.

Temporal Joins

Say that we want to derive the traditional temporal join on the history of salary and title temporal join, to derive the state-oriented representation with tuple-level time stamping shown in Figure 5. As illustrated by Figure 4, the temporal join can be computed by finding the overlapping segments of the history of title and department. These can be easily specified after we project out the **salary** column from the table of Figure 3, and eliminate the **salary** events using the condition **Type<>1**. If **EmpTD** denotes the tuple resulting from this selection/projection operation, then the temporal join can be derived using the query shown in 6. The temporal join can be expressed by finding the sequences that satisfy PATTERN**(A B+)**, where **B+** denote one or more events that have a different type and overlapping periods with respect to **A**. Thus, with our events ordered by their **start** time, we seek a new title (resp. department) event that is followed immediately by one or more department (resp. title) events. Consider for instance the histories of Figure 4. The first event is T_1; but once **A** is bound to this event, the event that follows immediately is T_2; now T_2 fails to satisfy **B+** because it violates the condition **A.Type<>B.Type**. Thus, the search restarts by matching **A** with T_2 and **B+** with D_1 D_2 D_3; as result of this successful match, the SELECT statement in our query returns the three segments of T_2 that overlap with D_1 D_2 D_3. (Example 6 uses a binary function **ets** that returns the earliest of two time stamps, and the binary function **nnv** that returns its first argument if this is not null, and the second one otherwise (both functions can be expressed by SQL CASE statements[2]). Our search for pattern now resumes and the next successful match binds **A** with D_3 and **B+** with T_3 T_4, returning their overlapping segments in symmetric fashion with respect to the previous match.

Example 6: From **EmpUH** to **EmpTD**: Temporal Join

```
SELECT Empno, B.start, ets(A.end, B.end),
    nnv(A.Title, B.Title), nnv(A.deptno, B.deptno)
FROM EmpTD AS PATTERN(A B+)
PARTITION BY Empno ORDER BY start, Type
WHERE A.Type <> B.Type
```

Either query will produce the result shown in Figure 5.

Temporal Coalescing

In addition to temporal joins, PATTERN SQL can also express effectively temporal coalescing. Say, for instance that we project out **Title** from our table **EmpTD** of Figure 5.

As illustrate by Figure 5, we can compute a coalesced interval by finding patterns where next interval has a **start** timestamp that precedes the current max of the **end** timestamps. Therefore, we have the following query:

Example 7: Coalescing Expressed using Kleene-closure

```
SELECT empno, first(B.start), max(B.end)
FROM EmpTD AS PATTERN (B+)
PARTITION BY empno ORDER BY start
WHERE count(B.*)=1
OR B.start<=max(previous(B.end))
```

[2]E.g., **ets(A.end, C.end)** can be expressed as: CASE WHEN **A.end** < **C.end** THEN **A.end** ELSE **C.end** END. Also, while **nnv** behaves exactly as SQL **coalesce** we did not use the latter to avoid naming confusions.

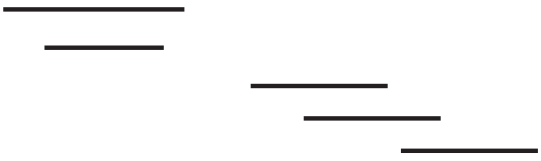

Fig. 6. Five Periods Needing Coalescing

The pattern **B+** denotes one or more intervals, provided that these satisfy the conditions in the WHERE. Now, take the five intervals shown in Figure 6, where they are arranged by their **start** time. In a group of overlapping interval, the first always satisfies the pattern through the condition **count(B.*)=1**. The intervals that follow the first one must instead satisfy the condition **B.start<=max(previous(B.end))** which checks that the new interval starts no later than the rightmost end of the intervals encountered so far (i.e., that no hole occurs after this interval). For the five periods in Figure 6, we see that this condition fails for the third interval, and therefore, the coalesced result of the first two intervals is returned. Now the pattern matcher resumes with the next group containing the rightmost three intervals, which are coalesced in a similar fashion.

An obvious observation that can be made about the previous temporal coalescing and the temporal join examples is that the computations needed to match their Kleene-closure patterns against the data are very similar to those that a procedural programmer would use to implement these functions on ordered sequences. This suggests that temporal queries expressed using pattern extensions of SQL are conducive to efficient implementation.

V. Conclusions

The examples discussed in the previous sections show that Kleene-closure constructs being proposed for SQL extend its power and flexibility in expressing temporal queries. Indeed, in addition to find complex event patterns, we can now express the derivation of temporal joins, and coalescing. However, to be fully effective, this approach requires suitable views, such as the unified view of Figure 3, which represent the history of atomic events (such as a new salary or a new department for our employee) along with the period during which the effect of each event remains valid. This unified view supports well complex queries, including coalescing and temporal joins. However, consider again our query *"What were the salary and title of employee 1001 on 1995-09-15?"*. To express this query against the unified view of Figure 3, we need simple selection operations to extract start, end and Type, salary and the title at time 1995-09-15. But an additional natural join operation is needed to combine them into one tuple. While this is a regular, non-temporal join, and thus would represent only a minor complication for current SQL users, one might also consider providing the representation of Figure 1 as an additional external view. Indeed, while the internal model of our transaction-time database is event-based, external views featuring both state-oriented and event-oriented features can be easily provided to facilitate the expression of temporal queries. In fact, much progress has already been made toward providing internal representations that are supportive of the different temporal views and queries discussed in this paper. At UCLA, we have developed an efficient transaction-time temporal database system ArchIS [44], we have then extended it into the PRIMA DBMS that supports schema evolution [19] by query rewriting. ArchIS and PRIMA extend the physical support provided by current DBMS with enhancements that enable temporal indexing and clustering. The internal storage model used in our representation is largely based on the H-table representation of Figure 2. Among the external views currently supported in our system, we have XML views whereby temporal queries can be expressed in XQuery, that our systems translate into equivalent SQL/XML statements. Work in progress focuses on supporting the unified views of Figure 3 on top of our current internal representation. This work involves the re-engineering of our current SQL-TS implementation for the Stream Mill Data Stream Management System (DSMS) [45], [46]. Indeed, while the same Klenee-closure SQL extensions have been proven to be very effective on both DSMS and traditional databases, different implementation and query optimization techniques are required for DBMS. Even so, the fact that the same queries that were developed to find useful patterns on data from the past can now be used to search data streams in real time represents a very interesting development.

Acknowledgments

The author is grateful to Barzan Mozafari, Carlo Curino, Nikolay Laptev, and the reviewers for their comments and suggested improvements.

References

[1] G. Ozsoyoglu and R.T. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, 1995.
[2] R. T. Snodgrass. Temporal Object-Oriented Databases: a Critical Comparision. *Addions-Wesley/ACM Press*, 1995.
[3] F. Grandi. An Annotated Bibliography on Temporal and Evolution Aspects in the World Wide Web. In *TimeCenter Technique Report*, 2003.
[4] R. T. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer, 1995.
[5] David Toman. Point vs. interval-based query languages for temporal databases. In *PODS '96*, pages 58–67, New York, NY, USA, 1996. ACM.
[6] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner. Transitioning Temporal Support in TSQL2 to SQL3. *Lecture Notes in Computer Science*, 1399:150–194, 1998.
[7] Krishna Kulkarni. System-versioned Tables–SQL Change Proposal. 2008.
[8] Fred Zemke, Andrew Witkowski, Mitch Cherniak, and Latha Colby. Pattern matching in sequences of rows,[sql change proposal, march 2007]. http://asktom.oracle.com/tkyte/row-pattern-recogniton-11-public.pdf http://www.sqlsnippets.com/en/topic-12162.html, 2007.
[9] The Tom Kyte Blog. So, in your opinion ... a sql feature under consideration.... pattern matching over partitioned, ordered rows in sql. http://tkyte.blogspot.com/2007/04/so-in-your-opinion.html, 2007.
[10] C. S. Jensen and R. T. Snodgrass. Temporal query languages. In *Temporal Database Entries for the Springer Encyclpedia of Database Systems*, volume Time Center Technical Report TR-90, 2008.
[11] Database Languages SQL, ISO/IEC 9075-*:1992.
[12] Database Languages SQL, ISO/IEC 9075-*:2003.
[13] A. Eisenberg, J. Melton, K. Kulkarni, J. Michels, and F. Zemke. Sql:2003 has been published. *SIGMOD Rec.*, 33(1):119–126, 2004.

[14] SQL 2003 Standard Support in Oracle Database 10g, otn.oracle.com/products/database/application_development/pdf/SQL_2003_TWP.pdf.

[15] C. S. Jensen and R. T. Snodgrass. Transaction time. In *Temporal Database Entries for the Springer Encyclpedia of Database Systems*, volume Time Center Technical Report TR-90, 2008.

[16] David B. Lomet and Feifei Li. Improving transaction-time dbms performance and functionality. In *ICDE*, pages 581–591, 2009.

[17] M. M. Moro and V. J. Tsotras. Transaction-time indexing. In *Temporal Database Entries for the Springer Encyclpedia of Database Systems*, volume Time Center Technical Report TR-90, 2008.

[18] Oracle Flashback Technology. http://otn.oracle.com/deploy/availability/htdocs/flashback_overview.htm.

[19] Hyun J. Moon, Carlo Curino, Alin Deutsch, Chien-Yi Hou, and Carlo Zaniolo. Managing and querying transaction-time databases under schema evolution. *PVLDB*, 1(1):882–895, 2008.

[20] F. Grandi and F. Mandreoli. The Valid Web: An XML/XSL Infrastructure for Temporal Management of Web Documents. In *ADVIS*, 2000.

[21] M. Gergatsoulis and Y. Stavrakas. Representing Changes in XML Documents using Dimensions. In *Xsym*, 2003.

[22] T. Amagasa, M. Yoshikawa, and S. Uemura. A Data Model for Temporal XML Documents. In *DEXA*, 2000.

[23] C.E. Dyreson. Observing Transaction-Time Semantics with TTXPath. In *WISE*, 2001.

[24] S. Zhang and C. Dyreson. Adding Valid Time to XPath. In *DNIS*, 2002.

[25] D. Gao and R. T. Snodgrass. Temporal Slicing in the Evaluation of XML Queries. In *VLDB*, 2003.

[26] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence query processing. In Richard T. Snodgrass and Marianne Winslett, editors, *SIGMOD*, pages 430–441. ACM Press, 1994.

[27] Praveen Seshadri and Arun N. Swami. Generalized partial indexes. In *Proceedings of Eleventh International Conference on Data Engineering 1995*, pages 420–427. IEEE Computer Society, 1995.

[28] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. Beyer, and M. Krishnaprasad. Srql: Sorted relational query language, 1998.

[29] Chang-Shing Perng and D. S. Parker. SQL/LPP: A time series extension of SQL based on limited patience patterns. In *DEXA*, volume 1677 of *Lecture Notes in Computer Science*. Springer, 1999.

[30] Reza Sadri, Carlo Zaniolo, Amir Zarkesh, and Jafar Adibi. Optimization of sequence queries in database systems. In *PODS*, Santa Barbara, CA, May 2001.

[31] Reza Sadri, Carlo Zaniolo, and Amir M. Zarkesh andJafar Adibi. A sequential pattern query language for supporting instant data minining for e-services. In *VLDB*, pages 653–656, 2001.

[32] Reza Sadri, Carlo Zaniolo, Amir M. Zarkesh, and Jafar Adibi. Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Syst.*, 29(2):282–318, 2004.

[33] Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker M. White. Towards expressive publish/subscribe systems. In *EDBT*, pages 627–644, 2006.

[34] Lilian Harada and Yuuji Hotta. Order checking in a cpoe using event analyzer. In *CIKM*, pages 549–555, 2005.

[35] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *SIGMOD Conference*, pages 147–160, 2008.

[36] Xin Zhou, Fusheng Wang, and Carlo Zaniolo. Efficient temporal coalescing query support in relational database systems. In *DEXA*, 2006.

[37] Byoung-Kee Yi, H. V. Jagadish, and Christos Faloutsos. Efficient retrieval of similar time sequences under time warping. In *ICDE*, pages 201–208, 1998.

[38] Michail Vlachos, Dimitrios Gunopulos, and George Kollios. Discovering similar multidimensional trajectories. In *ICDE*, pages 673–684, 2002.

[39] Lei Chen 0002, M. Tamer Özsu, and Vincent Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD Conference*, pages 491–502, 2005.

[40] Hanghang Tong, Yasushi Sakurai, Tina Eliassi-Rad, and Christos Faloutsos. Fast mining of complex time-stamped events. In *CIKM*, pages 759–768, 2008.

[41] Faisal I. Bashir, Ashfaq A. Khokhar, and Dan Schonfeld. Object trajectory-based activity classification and recognition using hidden markov models. *IEEE Transactions on Image Processing*, 16(7):1912–1919, 2007.

[42] Jae-Gil Lee, Jiawei Han, Xiaolei Li, and Hector Gonzalez. *raClass*: trajectory classification using hierarchical region-based and trajectory-based clustering. *PVLDB*, 1(1):1081–1094, 2008.

[43] C. Zaniolo, S. Ceri, C. Faloutsos, R.T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems*. Morgan Kaufmann, isbn 1-55860-443-x edition, 1997.

[44] Fusheng Wang, Carlo Zaniolo, and Xin Zhou. Archis: an xml-based approach to transaction-time temporal database systems. *VLDB J.* 17(6): 1445-1463 (2008)

[45] Y. Bai, C. Luo, H. Thakkar, and C. Zaniolo. Efficient support for time series queries in data stream management systems. In *Stream Data Management–Chapter 6*. N. Chaudhry, K. Shaw and M. Abdelguerfi (EDs.), Kluwer, 2004.

[46] Y. Bai, H. Thakkar, C. Luo, H. Wang, and C. Zaniolo. A data stream language and system designed for power and extensibility. In *CIKM*, pages 337–346, 2006.