

# Universal Temporal Data Languages

Cindy Xinmin Chen and Carlo Zaniolo

Computer Science Department  
University of California at Los Angeles  
Los Angeles, CA 90095  
cchen@cs.ucla.edu zaniolo@cs.ucla.edu

**Abstract.** Temporal reasoning and temporal query languages present difficult research challenges, which are slowly yielding to the combined attack of many investigations motivated by the theoretical interest and practical import of the problem. In this paper, we subscribe to TSQL2 insofar as practical requirements for a query language are concerned, but we propose a solution that overcomes its shortcomings, particularly the lack of universality whereby TSQL2 temporal extensions can not be easily applied to other query languages, such as QBE and Datalog. In this paper, we use Datalog as a framework to develop a new language — Temporal Data Language (TDL). To support our claim of universality, we argue that TDL constructs and semantics can be directly applied to derive temporal extensions of languages, such as QBE and SQL. Finally, we evaluate alternative approaches to the implementation of TDL, using as the basis for implementation the  $\mathcal{LDC}++$  system with extended aggregates developed at UCLA.

## 1 Introduction

Extensive research has focused on temporal databases [Tan93]. [Cho93] addressed temporal query languages based on logic programs. [Sno95] introduced TSQL2, which is a consensus extension to SQL-92. TSQL2 uses a Bi-temporal Conceptual Data Model (BCDM) [Jen94]. With respect to interval-based models, TSQL2 presents the advantage that it does not require explicit coalescing of time periods [BSS96]. One drawback of TSQL2 is that time columns cannot be explicitly referred in the SELECT and WHERE clauses of an SQL query: thus, we say that *TSQL2 has an implicit-time data model*. Because of implicit time, the semantics of TSQL2 queries become difficult to formalize, and complex queries are hard to express and indeed require the introduction of special constructs. The second problem is its lack of universality: while [BCS96] proposed a possible unification between TSQL2 and temporal logic, TSQL2's approach is specific to SQL, and does not contain the kernel of a universal temporal data model and query language, which can be applied to, say, QBE and Datalog.

Point-based versus interval-based temporal data models were discussed in [Tom96] and [BBJ98]. Point-based data models are directly based on the snapshot representation and support automatic coalescing, thus yielding a simpler syntax and clearer semantics. In point-based models, intervals are treated as an abstraction of sets of points. Snapshot-equivalent argument relations are treated identically in point-based data models, thus avoiding the representation dependency problem of interval-based data models.

In this paper, we present a new temporal query language, the Temporal Data Language (TDL), which has a unified semantics and serves as a natural extension to SQL, QBE and Datalog. TDL draws a clear distinction between end-user view, conceptual level, and physical level. For conceptual schema, TDL basically subscribes to the point-based approach proposed

by [Tom97, Tom98]. Our main novelty, with respect to the cited works, is a simple and uniform approach to Allen's interval operators and other complex TSQL2 constructs. We also take the first step toward an efficient  $\mathcal{LDC}++$  implementation and the application of the same extensions to SQL and QBE.

Because of space limitations, we only discuss valid time in this paper, although TDL can also handle transaction time. To illustrate some of the issues with TSQL2, consider the following queries based on examples given in [ADS97].

## 1.1 TSQL2

We have a patient database with the history of prescriptions given to patients. The schema and sample TSQL2 queries are as follows:

### 1. Schema definition

**Example 1** *Define the Prescription relation*

```
CREATE TABLE Prescription (Name CHAR(30),
    Physician CHAR(30), Drug CHAR(30), Dosage CHAR(30),
    Frequency INTERVAL MINUTE)
AS VALID STATE DAY
```

The Prescription relation is a valid time relation. The valid time has a granularity of one day.

### 2. Temporal selection and join

**Example 2** *What drugs have been prescribed with Proventil?*

```
SELECT P1.Name, P2.Drug
FROM Prescription AS P1, Prescription AS P2
WHERE P1.Drug = 'Proventil' AND P2.Drug <> 'Proventil'
AND P1.Name = P2.Name
```

The query returns the patient's name, the drug and the maximal periods during which both that drug and Proventil were prescribed to the patient.

Undoubtedly, the expression of simple selection and joins represents the best feature of TSQL2, insofar as these queries remain the same as in standard SQL. This is accomplished by keeping the time dimension implicit, as illustrated by the fact that the valid-time column is not even mentioned in the previous query. Therefore, by default, a TSQL2 query on a valid-time relation returns a valid-time relation. Additional constructs must then be used to deviate from this behavior. For instance, the keyword **snapshot** must be added to produce a normal relation instead of a valid-time one; while adding the keyword **snapshot** represents a simple modification, other changes discussed next are not so simple.

### 3. VALID clause

The VALID clause is used to override the default timestamp of the resulting tuple of a query.

**Example 3** *What drugs were Melanie prescribed during 1996?*

```
SELECT Drug
VALID INTERSECT(VALID(Prescription), PERIOD '[1996]' DAY)
FROM Prescription
WHERE Name = 'Melanie'
```

The query returns drugs, if any, prescribed to Melanie in 1996 and the maximal periods during which Melanie took the drugs. There will be tuples returned if some drugs were prescribed in 1996, but rather than the whole drug history being shown, only the history for the 1996 period will be returned. The need for this special construct **valid** is created by the fact that time in TSQL2 is kept implicit.

### 4. Restructuring and Allen's interval operators

An obvious requirement of all temporal languages is to support Allen's interval operators such as **overlaps**, **precedes**, **contains**, **equals**, **meets**, and **intersects** [All83].

Temporal languages that are based on temporal intervals [LM97] rely on these operators to express temporal joins. In this kind of languages, the query of Example 2 would be expressed by the condition 'P1 OVERLAPS P2'. No explicit use of **overlaps** is needed in point-based semantics since interval overlapping tantamounts to equality between some of their points [Tom96, BBJ98]. TSQL2 follows the point-based semantics, but assumes that the equality between time points holds implicitly (rather than asking the user to explicitly state it in the query). TSQL2, however, must provide explicit constructs for the remaining Allen's operators. For instance:

**Example 4** *Find the patients who have been prescribed Proventil, provided that all such prescription periods are in 1996.*

```
SELECT SNAPSHOT Name
FROM Prescription(Name, Drug) AS P
WHERE P.Drug = 'Proventil'
      AND CONTAINS(PERIOD '[1996]' DAY, VALID(P))
```

The query returns the patient's name if the patient took Proventil during 1996 and never before or after 1996.

Consider Prescription(Name, Drug) **AS P** in the FROM clause. This illustrates another TSQL2's "special" construct called *restructuring*. The function of this construct is to perform temporal projection with coalescing on the selected relations. The meaning of VALID(P) and **contains** in the WHERE clause is implicitly determined by this restructuring. Say for instance that the FROM clause becomes: Prescription(Name, Drug, Physician) **AS P**; then the query will return the maximal periods where Proventil was prescribed by the same physician. A patient, who was prescribed Proventil in March

1996 by physician A, would satisfy the query as long as he was never prescribed Proventil again in his life by A, even if he was prescribed Proventil by other physicians. However with Prescription(Name, Drug) AS P, the prescriptions by different physicians of the same drug to the same person are coalesced together.

## 5. Partitioning

This is yet another “special” new construct introduced by TSQL2. Syntactically it can be expressed by appending the **period** construct to a restructured projection in the FROM clause. Semantically, partitioning deals with the fact that TSQL2’s temporal elements are sets of periods, not single periods. Thus in the previous query a patient who took Proventil in October 1988 and March 1996 does not qualify — insofar as

**CONTAINS(PERIOD ‘[1996]’ DAY, VALID(P))**

requires that all the periods must be contained in 1996. The keyword **period** transforms a tuple which is valid in  $n$  periods to  $n$  tuples each valid in one of those periods.

**Example 5** *Find the patients who have been prescribed Proventil for one or more (continuous) periods within 1996.*

```
SELECT SNAPSHOT Name
FROM Prescription(Name, Drug) (PERIOD) AS P
WHERE P.Drug = ‘Proventil’
      AND CONTAINS(PERIOD ‘[1996]’ DAY, VALID(P))
```

The query returns names of patients who took Proventil for some continuous period that began and ended in 1996. For instance, a patient who took Proventil in March 1996, and then again in October 1997, satisfies this query, but not the previous query of Example 4. A patient who was continuously on Proventil from March 1996 till April 1997 does not satisfy this query. Here is another example of partitioning.

**Example 6** *Who has been on Proventil for a period of more than six consecutive months?*

```
SELECT SNAPSHOT Name
FROM Prescription(Name, Drug)(PERIOD) AS P
WHERE CAST(VALID(P) AS INTERVAL MONTH)
      > INTERVAL ‘6’ MONTH
```

The query returns the patient’s name if the patient has taken Proventil for more than six consecutive months. Partitioning allows the intermediate relation to have value-equivalent tuples timestamped with different maximal periods (thus violating the BCDM data model).

This paper is organized as follows: Section 2 introduces a new approach of querying temporal databases. In Section 3 the Temporal Data Language (TDL) is introduced as an extension of  $\mathcal{LDL}++$  [Zan88, NT89]; the previous six TSQL2 examples are then reexpressed in TDL. Then, Section 4 discusses the semantics and universality for TDL, and Section 5 proposes various implementation approaches for TDL. Section 6 concludes the paper.

## 2 A New Approach

The previous examples provide a simplified sample of the many and complex queries that occur in the framework of temporal databases. TSQL2's ease of specifying projections and joins (Example 2) represents a very desirable feature, but the many additional constructs needed to support other kinds of queries (e.g., those involving Allen's interval operators other than **overlaps** — Examples 3–6) represent an important drawback of the language.

The shortcomings of TSQL2 become obvious if one considers different syntactic frameworks. Many relational language frameworks proposed in the past, such as SQL, QBE, QUEL and Datalog, all shared a common logic-based semantics that allow natural translation of constructs developed in one framework to the others. For instance, since the experience of SQL, QBE and QUEL has well demonstrated that set aggregates should be included as primitive in a query language, one of the first tasks undertaken by the designers of  $\mathcal{LDC}$  was extending Datalog with set aggregates [ZAO93]. Conversely, the experience gained with recursive queries in Datalog has recently resulted in the introduction of new recursive query constructs for SQL [FMM96]. TSQL2 offers no hope for such cross-fertilization because its new temporal constructs, (restructuring, partitioning, etc.) are totally idiosyncratic to SQL. In the introduction, we have referred to these limitations as *lack of universality*. Moreover, the TSQL2 approach is particularly crippling for the Datalog language and its elegant formal semantics, whereby the meaning of a program is constructed from the symbols appearing in the program. It is hard to see how Datalog semantics can be salvaged in a query language where time remains implicit rather than being a part of the query program.

Therefore, our approach to temporal extensions will be based on explicit time. To represent explicit time, we adopt a point-based representation at the conceptual level. As shown in [Tom97, Tom98], this representation offers the important advantage that select-project-join queries are expressed in a very natural way, without requiring the (implicit-or-explicit) coalescing demanded by interval-based temporal languages.

Our point-based temporal model assumes: (i) a time granularity is used, e.g., days; (ii) the tuples in the temporal relation contain an additional column, say the last column, where a single time-granule is stored; and (iii) each temporal relation contains one tuple for each (time) point at which the database fact is valid.

In Datalog, rather than using tuples in database relations, we will use database facts. The simplest kind of temporal fact we can consider is a temporal propositional predicate — i.e., a predicate whose only column is the temporal column. For instance, assuming that our valid time has granularity of one day, and is represented by terms `date(Year, Month, Day)`, the propositional temporal predicate `year_1996` could be represented by the following sets of 366 facts:

```
year_1996(date(1996, 1, 1)).
year_1996(date(1996, 1, 2)).
. . .
year_1996(date(1996, 12, 31)).
```

We will use this point-based representation at the conceptual level — i.e., to define the syntax and semantics of the Temporal Data Language (TDL), which is simple, expressive, and universal.

In terms of implementation, efficiency considerations demand that we work with more succinct temporal representations. An interval-based representation, and an event-based

representation constitute two natural candidates for the tasks; these will be studied in Section 5. Therefore, our proposal takes full advantage of the three level DB architecture proposed by ANSI/SPARC, with an internal representation, a conceptual one, and an external one.

Our approach is different from Toman's [Tom97, Tom98] approach in several respects. Our main advance (w.r.t. Toman's approach) is that TDL uses *aggregates* to capture and express naturally all new complex temporal constructs used by TSQL2, including restructuring, partitioning, and Allen's temporal operators [All83].

In summary, the defining features of our new approach are:

1. Explicit valid time
2. Point-based representation
3. No additional construct (other than new temporal aggregates)
4. Efficient implementation by mapping point-based data model into equivalent internal representations.

The universality of our approach follows directly from the third point above. All existing languages already contain aggregate constructs, and they can easily accommodate new aggregates within their existing syntax, e.g., user-defined aggregates are part of the new SQL3 standards.

### 3 Temporal Data Language (TDL)

TDL extends Datalog by adding an explicit temporal element. One of the basic temporal data types supported in TDL is DATE: time-point with granularity of days, which is basically the same as SQL-92's *datetime* [DD93]. In TDL, DATE is a complex term which consists of three integers — “YEAR”, “MONTH” and “DAY”, whose values are constrained by the rules of the Gregorian Calendar.

We now express the previous TSQL2 examples in TDL:

#### 1. Schema declaration

**Example 7** *Define the Prescription relation (same as Example 1)*

```
database({prescript(Name : string, Physician : string, Drug : string,
                   Dosage_mg : integer, Frequency_Minute : integer,
                   Day : DATE)}).
```

The first five fields in the database schema are the same as those in TSQL2. An explicit temporal element, “Day”, which has a type of DATE, is added and represents the valid time of the database fact. We use the granularity of one day in the sample queries.

#### 2. Temporal selection and join

**Example 8** *What drugs have been prescribed with Proventil (same as Example 2)?*

```

query_2(Name, Drug, Day) ← prescript(Name, -, 'Proventil', -, -, Day),
                           prescript(Name, -, Drug, -, -, Day),
                           Drug ≈= 'Proventil'.

```

This query returns the patient's name, the drug and the date when both that drug and Proventil were prescribed to the patient. In TDL, temporal join is performed by specifying the same argument name for the temporal elements ("Day" in this case) in the joining predicates of the query. The symbol '≈=' denotes inequality.

Thus, in TDL, temporal selection and projection are trivial. Temporal join is done by explicitly specifying that the two joining relations have the same temporal element.

### 3. Valid clause

**Example 9** *What drugs were Melanie prescribed during 1996 (same as Example 3)?*

```

query_3(Drug, Day) ← year_1996(Day),
                    prescript(Name, -, Drug, -, -, Day),
                    Name = 'Melanie'.

```

The query returns drugs, if any, prescribed to Melanie in 1996. On the other hand, if we want the complete history of Melanie's drug prescription, the goal `year_1996(Day)` will be removed from the rule. Therefore, TDL's explicit time makes it easy to control both the conditions and the target time.

### 4. Restructuring and Allen's temporal operators

**Example 10** *Find the patients who have been prescribed Proventil, where all the prescription intervals are within the year 1996 (same as Example 4).*

```

query_4(Name, contains < (Day1, Day2) >) ← year_1996(Day1),
                                           prescript(Name, -, Drug, -, -, Day2),
                                           Drug = 'Proventil'.

```

Here `contains <>` is an  $\mathcal{LDC}++$  aggregate that behaves as a boolean operator: it returns a zero-arity tuple `()` for each pair of sets of values `t1` and `t2` if the set of `t1` contains the set of `t2`. Thus we obtain a set of tuples `(Name, ())` for each value of `Name` for which the subset relationship is satisfied, while other values of `Name` for which the relation is not satisfied are simply dropped.

Unlike TSQL2 which needs to use restructuring to perform coalescing on selected relations, TDL uses explicit time and aggregates to identify the columns on which the time points are grouped into maximal periods.

TDL uses user-defined aggregates to implement Allen's temporal operators. In addition to `contains`, the other Allen's operators supported in TDL via aggregates include `precedes`, `equals`, `meets`; there is no `overlaps` and `intersects` aggregate since these are automatically supported by the point-based semantics. Thus we have the following:

- The aggregate `contains<(Day1, Day2)>` is satisfied if every `Day2` date is also a `Day1` date.

- The aggregate precedes<(Day1, Day2)> is satisfied if all the Day1 dates precede the Day2 dates — i.e., if the max of the former set of dates precedes the least of the latter set of dates.
- The aggregate equals<(Day1, Day2)> is satisfied when both the conditions contains<(Day1, Day2)> and contains<(Day2, Day1)> are satisfied.
- The aggregate meets<(Day1, Day2)> is satisfied when the last Day1 date coincides with the first Day2 date, i.e, the max of the former set of dates is the same as the least of the later set of dates.

In reality, these operators support a generalization of Allen’s original logic because they apply to sets of periods rather than to a single period. In TSQL2, partitioning is used to translate a set of periods into individual periods that can be operated on separately.

## 5. Partitioning

**Example 11** *Find the patients who have been prescribed Proventil for one or more periods in 1996(same as Example 5).*

```
query_5(Name, ID, contains < (Day1, Day2) >) ← year_1996(Day1),
                                             prs_prd(Name, Drug, (ID, Day2)),
                                             Drug = 'Proventil'.
```

```
prs_prd(Name, Drug, period < Day >) ← prescript(Name, -, Drug, -, -, Day).
```

The aggregate period<Day> transforms a set of dates into a period by assigning each date an ID, which is an integer denoting whether this date belongs to the first, second, ...,  $n^{th}$  continuous segment in the sequence. All the dates associated with the same ID belong to the same segment. The use of the aggregate **period** performs partitioning while retains the point-based semantics of the TDL data model.

**Example 12** *Who has been on Proventil for a period of more than six consecutive months (same as Example 6)?*

```
query_6(Name) ← months(Name, Drug, ID, Months),
                Drug = 'Proventil', Months > 6.
```

```
months(Name, Drug, ID, length_months < Day >) ← prs_prd(Name, Drug, (ID, Day)).
```

The query returns the patient’s name if the patient has taken Proventil for more than six consecutive months. This query uses the **prs\_prd** defined in the previous example. The aggregate length\_months<Day> first forms the dates into a set, then counts the number of dates in the set, and returns the count in terms of months assuming each month has 30 days. Similar aggregates include length\_days<Day> and length\_years<Day> which return the count in terms of days and years, respectively, assuming each year has 365 days.

In the last three examples, we have constructed snapshot relations where we are only interested in **Names** disregarding their history. But if we wanted to find the patient’s

name and the days in those periods that satisfy the query, we would rewrite `query_6` as follows:

```
query_7(Name, Day) ← months(Name, Drug, ID, Months),
                    Drug = 'Proventil', Months > 6,
                    prs_prd(Name, Drug, (ID, Day)).
```

Finally, if we wanted the entire prescription history of the patients satisfying the previous query, then we would write:

```
query_8(Name, Day) ← months(Name, Drug, ID, Months),
                    Drug = 'Proventil', Months > 6,
                    prescript(Name, -, -, -, Day).
```

Since similar considerations apply to the queries of Examples 10 and 11, we conclude that explicit time with aggregates provide great flexibility and power in expressing temporal queries.

## 4 Semantics and Universality

From the previous examples, we conclude that TDL is a natural extension to Datalog. No additional language construct is needed other than aggregates, a construct that is already found in deductive database languages. The operations involving time periods a.k.a. sets of time instants, are done via new temporal aggregates. After we take a second look, we see that in fact the new temporal aggregates are based on the traditional SQL aggregates. Thus  $\text{contains}\langle X, Y \rangle$  simply denotes that the set of Y-values is a subset of the set of X-values. In terms of traditional SQL aggregates, this can be expressed as:

$$\text{count}(S_Y) = \text{count}(S_X \cap S_Y)$$

where  $S_X$  denotes the set of X-values and  $S_Y$  denotes the set of Y-values. Likewise,  $\text{precedes}\langle X, Y \rangle$  and  $\text{meets}\langle X, Y \rangle$  can simply be computed by comparing the extrema in the two sets. Therefore the aggregates used to define Allen's temporal operators in our model can be reduced to the familiar, vanilla flavored, aggregates in SQL. The newly named aggregates are nevertheless important because of their significance to the user, and because they enable more efficient implementations. Thus, they provide significant practical advantages beyond those offered by TSQL/TP [Tom97]. In general, the TDL approach offers two significant benefits:

- Since aggregates in Datalog have a standard declarative logic-based semantics [WZ98] (i.e., least-model and least-fix-point based) TDL inherits elegant formal semantics of these languages.
- All database query languages, such as SQL and QBE, have aggregate constructs. Thus, to support temporal extensions along the lines illustrated by TDL, no additional language construct need to be introduced other than some new aggregates.

Therefore, TDL achieves formal semantics and universality in a simple fashion.

## 5 Implementing TDL

In this section, we will discuss various implementations of TDL. We will focus on the mappings between the point-based relation and the interval-based and event-based relations. We assume that all the facts stored in the database have an ascending order in terms of their temporal elements. Currently, TDL only supports the Gregorian Calendar, further work is needed to support other calendars and user defined calendars.

### 5.1 Interval-based implementation

Using an interval-based relation to implement TDL solves the space and efficiency problems associated with the point-based data model. In interval-based relations, all tuples are time-stamped with an two time instants: one indicates the start-point and the other end-point of the interval.

In interval-based implementation, the database schema in Example 1 is implemented as follows:

```
database({prescript(Name : string, Physician : string, Drug : string,
                    Dosage_mg : integer, Frequency_Minute : integer,
                    Interval : (DATE, DATE))}).
```

Using the aggregate **period** described in the previous section, we can map a query on a point-based view to its interval-based implementation.

The following rules show how the mapping is performed. In the examples,  $q$  is a point-based relation and  $p$  is an interval-based relation.

#### 1. Map time points into intervals

$$p(X_1, \dots, X_n, \text{int}(\text{Start\_Day}, \text{End\_Day})) \leftarrow \text{aux\_1\_1}(X_1, \dots, X_n, \text{ID}, \text{Start\_Day}, \text{End\_Day}).$$
$$\text{aux\_1\_1}(X_1, \dots, X_n, \text{ID}, \text{min\_day} \langle \text{Day} \rangle, \text{max\_day} \langle \text{Day} \rangle) \leftarrow \text{aux\_1\_2}(X_1, \dots, X_n, (\text{ID}, \text{Day})).$$
$$\text{aux\_1\_2}(X_1, \dots, X_n, \text{period} \langle \text{Day} \rangle) \leftarrow q(X_1, \dots, X_n, \text{Day}).$$

The predicate `aux_1_2` first constructs a period from all the dates in the relation  $q$ . The predicate `aux_1_1` calculates the minimum and maximum of all the dates which belong to a same segment of the period, i.e., attached with the same ID. The pair of minimal and maximal dates then serve as the start and end points of an interval in the final interval-based relation  $p$ .

The aggregates **min\_day** and **max\_day** are yet two more temporal aggregates in TDL. They take input as a set of dates and find the minimal and maximal dates of the set, respectively.

The new aggregate **instant** is defined to map a query from its interval-based implementation back to the point-based relation.

## 2. Map time intervals into points

$$q(X_1, \dots, X_n, \text{instant} < \text{Interval} >) \leftarrow p(X_1, \dots, X_n, \text{Interval}).$$

The aggregate **instant** takes an interval as input, and returns all the dates within that time interval. For example,

$$q(\text{instant} < \text{int}(\text{date}(1996, 1, 1), \text{date}(1996, 1, 31)) > \leftarrow p(\text{int}(\text{date}(1996, 1, 1), \text{date}(1996, 1, 31))).$$

returns

$$\begin{aligned} & q(\text{date}(1996, 1, 1)). \\ & q(\text{date}(1996, 1, 2)). \\ & \quad \cdot \quad \cdot \quad \cdot \\ & q(\text{date}(1996, 1, 31)). \end{aligned}$$

## 5.2 Event-based implementation

An event-based implementation offers all advantages of interval-based implementations and it is closer to the semantics of the point-based data model because of its event nature. In an event-based relation, all tuples are attached with an event-type specification and a single time instant. The event-type specifies which type of event the tuple represents, namely, **INS** and **DEL**, which stand for insertion and deletion, respectively. A tuple with the same non-temporal value may appear twice in a database, i.e., one for insertion of the fact and one for deletion of the fact. If no deletion tuple is in the database, then the fact is assumed to hold true until a default time assigned by the system. An update of a database fact is performed as a modification of the time specified in its old deletion tuple followed by inserting a pair of insertion and deletion tuples with new values into the database. All tuples are stored in the database according to the ascending order of their timestamps.

In event-based implementation, the database schema in Example 1 is defined as:

$$\text{database}(\{\text{prescript}(\text{Name} : \text{string}, \text{Physician} : \text{string}, \text{Drug} : \text{string}, \text{Dosage\_mg} : \text{integer}, \text{Frequency\_Minute} : \text{integer}, \text{Event} : \text{string}, \text{Day} : \text{DATE})\}).$$

The aggregates **min\_day** and **max\_day** are used to map a query in a point-based data model to its event-based implementation and the aggregates **pair\_up** and **instant** are used to map a query from its event-based implementation back to the point-based data model. The following examples show how the mappings are performed. In the examples,  $q$  is a point-based relation and  $r$  is an event-based relation.

### 1. Map time points into events

$$r(X_1, \dots, X_n, \text{Event}, \text{min\_day} < \text{Day} >) \leftarrow q(X_1, \dots, X_n, \text{Day}), \text{Event} = \text{'INS'}.$$

$$r(X_1, \dots, X_n, \text{Event}, \text{max\_day} < \text{Day} >) \leftarrow q(X_1, \dots, X_n, \text{Day}), \text{Event} = \text{'DEL'}.$$

The insertion time of database fact is the minimal date of the set of dates in the relation  $q$  and the deletion time of a database fact is the maximal date.

## 2. Map events into time points

$$q(X_1, \dots, X_n, \text{instant} < \text{int}(\text{StartDay}, \text{EndDay}) >) \leftarrow \\ \text{aux\_2\_1}(X_1, \dots, X_n, (\text{StartDay}, \text{EndDay})).$$
$$\text{aux\_2\_1}(X_1, \dots, X_n, \text{pair\_up} < (\text{Event}, \text{Day}) >) \leftarrow r(X_1, \dots, X_n, \text{Event}, \text{Day}).$$

An interval is first constructed with the insertion and deletion time of the database fact specified as its start and end points via `aux_2_1`. A new aggregate **pair\_up** is introduced here. It pairs up the two tuples which represent the insertion and deletion of a database fact. The aggregate takes the event types and dates as input and returns pairs of dates, i.e., the insertion date and the deletion date which can also be treated as the start and end of an interval. The aggregate **instants** is then used to extract all the dates within the interval. If no deletion tuple of a fact exists, then the assumed “until changed”, i.e., until the default value.

The event-based implementation retains the advantages of the interval-based implementation and has a closer relationship with composite event specification languages and provides an opportunity to combine active rules with temporal databases [MZ97].

## 6 Conclusion

We have shown that TDL is a natural temporal extension of Datalog. The same approach is also applicable to SQL and QBE. We used a point-based data model at the conceptual level and took the point-based reasoning approach beyond that of TSQL/TP [Tom97, Tom98] by using aggregates to express queries reasoning on time periods.

TDL has the same standard declarative logic-based semantics as Datalog and provides a universal approach to solve the problem of extending conventional database query languages, such as SQL, QBE and Datalog, to express temporal queries.

We have showed two different implementation approaches of TDL, i.e., interval-based and event-based. Both of the implementation approaches overcome the space inefficiency problem the point-based data model may cause.

Due to lack of space, we only showed TDL with valid time supported in the paper. TDL’s support for bi-temporal reasoning will be discussed in future papers.

## Acknowledgements

The authors would like to thank Haixun Wang for his help with temporal aggregates, and Reza Sadri and the referees for their helpful comments.

## References

- [All83] J. F. Allen. Maintaining knowledge about temporal intervals. In *Communications of the ACM, Vol. 26, No. 11*, pages 832-843, 1983
- [BCS96] M. H. Bohlen, J. Chomicki, R. T. Snodgrass and D. Toman. Querying TSQL2 Databases with Temporal Logic. In *Proceedings of the 5th Conference on Extended Database Technology*, pages 325-341, 1996

- [BSS96] M. H. Bohlen, R. T. Snodgrass and M. D. Soo. Coalescing in Temporal Databases. In *Proceedings of the 22nd International Conference on Very Large Databases*, pages 180-191, 1996
- [BBJ98] M. H. Bohlen, R. Busatto and C. S. Jensen. Point- Versus Interval-based Temporal Data Models. In *Proceedings of the 14th International Conference on Data Engineering*, pages 192-200, 1998
- [Cho93] J. Chomicki. Temporal Databases. In *Proceedings of the 12th ACM Symposium on Principles of Database Systems*, pages 1-16, 1993
- [DD93] C. J. Date and H. Darwen. *A Guide to the SQL Standard, third edition*, Addison-Wesley, 1993
- [FMM96] S. J. Finkelstein, N. Mattos, I. S. Mumick and H. Pirahesh. Expressing Recursive Queries in SQL. ISO/IEC JTC 1/SC 21/WG 3 DBL:MCI Rep. X3H2-96-075, 1996
- [Jen94] C. S. Jensen, et al. A Consensus Glossary of Temporal Database Concepts. In *SIGMOD Record, Vol. 23, No. 1*, pages 52-64, 1994
- [LM97] N. A. Lorentzos and Y. G. Mitsopoulos. SQL Extension for Interval Data. In *IEEE Transactions on Knowledge and Data Engineering, Vol. 9, No. 3*, pages 480-499, 1997
- [MZ97] I. Motakis and C. Zaniolo. Temporal Aggregation in Active Database Rules. In *SIGMOD Record, Vol. 26, No. 2*, pages 440-451, 1997
- [NT89] S. Naqvi and S. Tsur. *A Logical Language For Data And Knowledge Bases*, Computer Science Press, 1989
- [Sno95] R. T. Snodgrass, et al. *The TSQL2 Temporal Query Language*, Kluwer, 1995
- [Tan93] A. Tansel, et al. *Temporal Databases: theory, design and implementation*, Benjamin/Cumming, 1993
- [Tom96] D. Toman. Point vs. Interval-based Query Languages for Temporal Databases. In *Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages, 58-67, 1996
- [Tom97] D. Toman. A Point-Based Temporal Extension of SQL. In *Proceedings of the 6th International Conference on Deductive and Object-Oriented Databases*, pages 103-121, 1997
- [Tom98] D. Toman. Point-Based Temporal Extensions of SQL and their Efficient Implementation. To appear in Proceedings of Dagstuhl Workshop on Temporal Databases, 1998
- [WZ98] H. Wang and C. Zaniolo. User-Defined Aggregates for Logical Data Languages. *submitted to DDLP'98*, 1998
- [Zan88] C. Zaniolo. Design and Implementation of a Logic Based Language for Data Intensive Applications. In *Proceedings of the International Conference on Logic Programming*, 1988
- [ZAO93] C. Zaniolo, N. Arni and K. Ong. Negation and Aggregates in Recursive Rules: the LDL++ Approach. In *Proceedings of the 3rd International Conference on Deductive and Object-Oriented Databases*, pages 204-221, 1993
- [ADS97] C. Zaniolo, S. Ceri, C. Faloutsos, R. Snodgrass, and R. Zicari, *Advanced Database Systems*, Morgan Kaufmann, 1997