Universal Temporal Extensions for Database Languages

Cindy Xinmin Chen and Carlo Zaniolo
Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90095
E-mail: {cchen, zaniolo}@cs.ucla.edu

Abstract

Temporal reasoning and temporal query languages present difficult research problems of theoretical interest and practical importance. One problem is the chasm between point-based temporal reasoning and interval-based reasoning. Another problem is the lack of robustness and universality in many proposed solutions, whereby temporal extensions designed for one language cannot be easily applied to other query languages— e.g., extensions proposed for SQL cannot be applied to QBE or Datalog. In this paper, we provide a simple solution to both problems by observing that all query languages support (i) single-value based reasoning and (ii) aggregate-based reasoning, and then showing that these two modalities can be naturally extended to support, respectively, point-based and interval-based temporal queries. We follow TSQL2 insofar as practical requirements are concerned, and show that its functionality can be captured by simpler constructs which can be applied uniformly to Datalog, QBE and SQL. Then, we show that an efficient implementation can be achieved by mapping into a different storage representation, and discuss a prototype built along these lines using the $\mathcal{LDL}$++ system with extended aggregates.

1. Introduction

With many applications requiring support for temporal databases, extensive research has focused on temporal queries and reasoning [12]. A critical issue in all these approaches is the choice of model used to represent valid time: for instance, many approaches represent valid time states by temporal intervals, while point-based models view the database as a sequence of snapshots [3, 13]. The major drawback of the interval-based data model is the need for coalescing time intervals when a projection is taken, whereas point-based data models are free from this problem. On the other hand, many temporal queries can be expressed naturally using intervals, e.g., using Allen's interval operators. Furthermore, intervals provide an efficient representation for the physical storage of the data, while point-based conceptual models must be mapped into different representations for storage efficiency.

The language TSQL2 [11] introduced a consensus extension to SQL-92 that supports a Bi-temporal Conceptual Data Model [6], which handles both valid time and transaction time. In TSQL2, there is no explicit time column in the relations, and valid time cannot be referred in the query as a tuple attribute; therefore, we say that *TSQL2 has an implicit-time data model.* By keeping time implicit, TSQL2 eliminates the need for a user to specify the coalescing of time periods [2]. However, implicit time makes the semantics of TSQL2 more obscure and difficult to formalize. Furthermore, the fact that one cannot work directly with time makes certain queries hard to express and requires the introduction of special temporal constructs to achieve the same goal indirectly. This bring us to the second problem, i.e., said special TSQL2 constructs are designed for SQL only, and do not generalize into a universal temporal data model and query language that can be used for, say, QBE and Datalog. We refer to this problem as the *lack of universality* of TSQL2.

In this paper, we first propose an approach based on a point-based explicit-time model and on temporal aggregates to support interval-based reasoning. We demonstrate the universality of this approach, by proposing parallel designs for SQL, QBE, and Datalog. Then, we tackle the problem of efficient implementation for these languages, by proposing two storage representations, one based on intervals, and the other on events, which avoid the space inefficiency problem of point-based representations. Queries on point-based representations at the conceptual level can be easily mapped into equivalent queries on interval-based or event-based representations at the implementation level. Because of space limitations, we only discuss valid time in this paper, although our approach can be extended to handle transaction time.

2. TSQL2

To illustrate some of the issues with TSQL2, consider a patient database with the history of prescriptions given to patients as in [17]. The schema and sample TSQL2 queries are as follows:

1. Schema definition

   Example 1 *Define the* Prescript *relation*

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

   The Prescript relation is a valid time relation. The valid time has a granularity of one day. The important observation to be made here is that valid time must be qualified via annotations, since it is not a column of the relation.

2. Temporal selection and join

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

   ```
   SELECT P1.Name, P2.Drug
   FROM Prescript AS P1 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, queries involving only selections, projections, and joins represent the best feature of TSQL2, insofar as these queries are the same as in standard SQL-92. 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 using the keyword snapshot adds little complexity to a query, other constructs needed to override TSQL2's defaults are neither simple nor user-friendly. The VALID clause discussed next is an example.

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(Prescript),
           PERIOD '[1996]' DAY)
   FROM Prescript
   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 to Melanie in 1996; but, due to the VALID annotation added to the SELECT clause, only the drug history for 1996 is shown, rather than the complete history. The need for this special construct valid is created by the fact that time in TSQL2 is kept implicit.

3. Explicit Time Queries

The basic approach we propose here is based on a point-based temporal data model and on explicit-time queries. Our point-based temporal data model assumes:

- the use of some granularity for representing valid time—for instance we will uses days in our examples,
- every temporal relation contains an additional column, say the last column, called VTime, storing single time-granules, and
- the relation contains one row for each (time) point at which the database fact is valid.

For instance, a temporal (virtual) relation that is supported by the system, the calendar relation, is as follows:

Calendar	Year	Month	Day	VTime
	...	...	...	...
	1996	September	24	24/09/1996
	...	...	...	...

Here, we display only one tuple as a sample of our calendar relation. Also observe that we represent valid-time dates using the day/month/year notation.

This calendar relation is not a stored object, it is a virtual view that provides a user-friendly QBE interface to calendar queries, such as the following that returns all days in September 1996:

Calendar	Year	Month	Day	VTime
	P.1996	P.September		P.

In as much as this calendar query is implemented by an internal calendar function, it exemplifies the main idea of our approach: select a data model that simplifies the expression of complex temporal queries, and rely on mapping to efficient internal representations for implementation. The idea of different representations at different levels has been long popularized by the ANSI/X3/SPARC architecture for DBMS [4]. Effective representations at the storage level are discussed in Section 7. Also at the end-user level, simple solutions exist to avoid the redundant printouts generated by point-based representations. For instance, rather than having the previous query generate 30 tuples, identical in the year and month columns, we can use the following display to present the results:

Result	Year	Month	VTime
	1996	September	01/09/1996
	...	...	...
	1996	September	30/09/1996

Since people are quite adept at filling-in the dots, this is a concise and unambiguous representation for much larger sets. The same sets could also be represented by a single tuple as follows:

(1996, September, 01/09/1996 . . . 30/09/1996)

This can either be viewed as an unnormalized tuple, where September 1996 is associated with the set of days 01/09/1996 . . . 30/09/1996, or as an interval-based representation of the same information. An interval-based representation is acceptable at the user level, and has some properties that make it an interesting (although perhaps not the best) candidate at the storage level, as it will be discussed later.

3.1. Schema definition in SQL^T

We now describe SQL^T, our valid-time extension of SQL-92, where explicit time is used in schema declarations and queries.

Example 4 *Define the* Prescript *relation*

```
CREATE TABLE Prescript (Name CHAR(30),
      Physician CHAR(30), Drug CHAR(30),
      Dosage CHAR(30),
      Frequency INTERVAL MINUTE, VTime DAY)
```

Thus, the valid time has become the last column in our relation. The reserved keyword VTime must be used to denote the name of the valid-time column—a relation can have at most one of these columns. Similar conventions apply to the schemas defined in QBE, or to Datalog languages, such as $\mathcal{LDL}$++ [16, 18]. The expressions of temporal selection and join queries in SQL^T, QBE^T and Datalog^T are straightforward and are shown next.

3.2. Temporal Selection and Join

Example 5 *What drugs have been prescribed with Proventil?*

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

Prescript	Name	...	Drug	...	VTime
	_name		Proventil		_vtime
	P._name		P._drug		P._vtime

Conditions
_drug ¬= Proventil

```
query1(Name, Drug, VTime) ←
    prescript(Name, _, 'Proventil', _, _, VTime),
    prescript(Name, _, Drug, _, _, VTime),
    Drug ~= 'Proventil'.
```

3.3. The VALID Clause

TSQL2's VALID clause is no longer needed since, in SQL^T, the target time span can be explicitly controlled by conditions in the WHERE clause.

Example 6 *What drugs was Melanie prescribed during 1996?*

```
SELECT P.Drug, P.VTime
FROM Prescript AS P, Calendar AS C
WHERE P.Name = 'Melanie' AND C.Year = 1996
   AND P.VTime = C.VTime
```

The same query can be expressed as follows in QBE^T and Datalog^T:

Calendar	Year	Month	Day	VTime
	1996			_vtime

Prescript	Name	...	Drug	...	VTime
	Melanie		P._drug		P._vtime

$\texttt{query2(Drug, VTime)} \leftarrow$
$\quad \texttt{calendar(1996, \_, \_, VTime)},$
$\quad \texttt{prescript('Melanie', \_, Drug, \_, \_, VTime)}.$

4. Interval-Oriented Reasoning

An important requirement of all temporal languages is to support Allen's interval operators such as overlaps, precedes, contains, equals, meets, and intersects [1].

Temporal languages that are based on temporal intervals [7] 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 two intervals overlap if and only if they share some common points [3, 13]. This conclusion also holds for TSQL2, where equality between time points is assumed as default condition when no other temporal condition is given. In TSQL2, however, the user must use explicit constructs to specify the remaining Allen's operators. For instance, consider the following query:

Example 7 *TSQL2 Query: find the patients who have been prescribed Proventil, throughout 1996.*

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

This query returns the patient's name if the patient took Proventil for the whole period of 1996. Thus VALID(P) denotes the valid time of tuples in P represented as one or more intervals (i.e., periods in TSQL2 terminology), and PERIOD '[1996]' DAY is simply a constructor of a time period starting January 1, 1996, and ending December 31, 1996.

If a patient took Proventil during several noncontiguous time periods, then, at least one of these intervals must contain the "Year 1996" time period. This brings out the important point that TSQL2 is really dealing with sets of intervals (a *time element* in TSQL2's terminology), rather than a single interval. For now, let us ignore this point (discussed in great length in Section 6) as if every drug were only prescribed during one interval. Then, consider

FROM Prescript(Name, Drug) AS P

The attribute-list (Name, Drug) is an instance of TSQL2's "special" extension of SQL-92 called *restructuring.* The purpose of this construct is to define the attributes on which the tuples must be coalesced. Indeed Proventil might have been prescribed to the same patient by different physicians, and with different dosage and frequency. If we remove the attributes (Name, Drug) from the FROM clause of the previous query, the meaning of the query is changed into: find the patients who have been prescribed Proventil, *by the same physician, with the same dosage and frequency*, throughout 1996; this is a much stricter condition than the original one. Therefore, to support this query, we need to project out the physician, dosage, and frequency columns, and coalesce the time intervals into maximal intervals of time during which the values of attributes (Name, Drug) remain unchanged.

Many queries that arise in the context of interval-oriented reasoning involve duration. For instance, we might want to find the names of the patients who have been prescribed some drugs for a total of more than 240 days. Again, we have to use restructuring to ensure that we accumulate the length of a prescription independent of physician, dosage and frequency (e.g., to ensure that patients do not circumvent the maximum prescription period limitations by changing physician).

Example 8 *TSQL2 Query: find the patients who have been prescribed some drug for more than 240 days.*

```
SELECT SNAPSHOT Name
FROM Prescript(Name, Drug) AS P
WHERE CAST(VALID(P) AS INTERVAL DAY)
      > INTERVAL '240' DAY
```

Again, VALID(P) defines one or more periods. In TSQL2, the terms period and interval denote, respectively, anchored and unanchored spans of time. Thus, in TSQL2, casting is needed to convert a set of (one or more) periods to an interval by adding up the length of each period. Therefore, the query in Example 8 adds up the lengths in days of all periods during which a patient took the same drug, and checks whether the accumulated length exceeds 240 days. A more complex TSQL2 query is needed to find patients who took the same drug for more than 240 *consecutive* days—i.e., a continuous prescription of the same drug for more than 240 days; this is discussed in Section 6.

5. Temporal Aggregates

In a point-based temporal model, intervals are sets of contiguous points; thus *set aggregates* should be used to support interval-oriented reasoning. For instance, the last query can be formulated and expressed in SQL-92 as follows:

Example 9 *SQL-92 Query for Example 8*

```
SELECT Name
FROM Prescript
GROUP BY NAME, DRUG
HAVING COUNT(VTime) > 240
```

Observe that, unlike TSQL2 which had to introduce restructuring, no new construct is needed here. The set of attributes each period is associated with is specified explicitly and unequivocally by the group-by attributes NAME, DRUG.

While the semantics of the count aggregate faithfully expresses the concept of duration, we will introduce in SQL^T a special temporal aggregate length to optimize:

- users' convenience of having mnemonic constructs to express the intuitive meaning of the intended operation (also we include versions that convert to different granularities, e.g., length_month to convert to numbers of months), and
- efficiency of execution since the implementation can be optimized directly from the storage representation used for valid time (e.g., an interval-based representation).

Therefore, in our SQL^T language, the previous query will be expressed as follows:

Example 10 *SQL^T Query for Example 8*

```
SELECT Name
FROM Prescript
GROUP BY NAME, DRUG
HAVING LENGTH(VTime) > 240
```

Different styles of aggregate queries are possible for SQL^T. For instance, we can express the query in Example 10 using nested sub-queries, as follows:

Example 11 *Temporal Aggregates in SQL-92 using Nested Sub-queries (same as Example 10)*

```
SELECT P.Name
FROM Prescript AS P
WHERE LENGTH (
      SELECT P1.VTime
      FROM Prescript AS P1
      WHERE P1.NAME = P.NAME
      AND P1.DRUG = P.DRUG) > 240
```

This second form is in fact preferable, since it is more general and accommodates any number of group-by combinations—unlike the explicit group-by form used in Example 10.

Since all *query languages support aggregates*, our new temporal aggregates can be added on without perturbing the syntactic and semantic structure of the original languages. Thus, in QBE^T, our query can be expressed as follows:

Example 12 *QBE^T Query for Example 8*

Prescript	Name	···	Drug	···	VTime
	P.G._name		G._drug		_vtime

Conditions
LENGTH._vtime > 240

Of particular interest, is the "G" appearing in the first and third columns of the first table; this denotes that _name and _drug serve as group-by columns for other variables, such as _vtime, that appear in the same row without "G" [9].

For Datalog^T languages, we will use the head-aggregation syntax of $\mathcal{LDL}$++ [19]. Then, our previous query can be expressed as follows:

Example 13 *Datalog^T Query for Example 8*

```
groupdays(Name, Drug, length⟨VTime⟩) ←
            prescript(Name, _, Drug, _, _, VTime).
query3(Name) ←  groupdays(Name, _, TotalDays),
                TotalDays > 240.
```

Here an attribute name followed by the pointed brackets denotes an aggregate column. *Every aggregate column is implicitly grouped by all the non-aggregate columns in the head of the rule.* Thus, in our example, we compute the aggregate length of VTime with respect to the two other columns Name and Drug.

In general, the limitations caused by implicit group-by attributes are easily overcome given the great flexibility of user-defined aggregates in $\mathcal{LDL}$++, Version 5 [16, 18]. In particular, we have defined a binary aggregate called contains which is true when one set of time points contains all the time points in the other set. Thus our query of Example 7 can be formulated as follows:

Example 14 *Datalog^T Query for Example 7: find the patients who have been prescribed Proventil, throughout 1996.*

```
query4(Name, contains⟨(VTime1,VTime2)⟩) ←
   prescript(Name,_,'Proventil',_,_,VTime1),
   calendar(1996,_,_,VTime2).
```

In an interval-based representation, VTime1 and VTime2 are implemented as two sets of intervals. Then, contains⟨(VTime1, VTime2)⟩ is implemented by checking that, for each interval I_2 in VTime2, there is an interval I_1 in VTime1, such that I_1 contains I_2.

The binary aggregate contains can also be used in QBE^T quite naturally:

Example 15 *QBE^T Query for Example 7*

Prescript	Name	⋯	Drug	⋯	VTime
	P.G._name		Proventil		_vtime1

Calendar	Year	Month	Day	VTime
	1996			_vtime2

conditions
CONTAINS.(_vtime1, _vtime2)

In SQL^T, the same query is expressed most naturally using the nested sub-query technique discussed previously:

Example 16 *SQL^T Query for Example 7*

```
SELECT P.Name
FROM Prescript AS P
WHERE
      ((SELECT P1.VTime
      FROM Prescript AS P1
      WHERE P1.Name = P.Name
         AND P1.Drug = P.Drug
         AND P1.Drug = 'Proventil')
      CONTAINS
      (SELECT C.VTime
      FROM Calendar AS C
      WHERE C.Year = 1996))
GROUP BY P.Name
```

The semantics of our new temporal aggregates can be defined from existing SQL-92 aggregates in a very natural fashion. For instance, CONTAINS(S1, S2) can be defined as a shorthand of

$$\mathrm{COUNT}(S1) = \mathrm{COUNT}(S1 \cap S2)$$

where the set intersection can be expressed using joins. Likewise, PRECEDES(S1, S2) and MEETS(S1, S2) can be respectively defined as $\mathrm{MAX}(S1) < \mathrm{MIN}(S2)$ and $\mathrm{MAX}(S1) = \mathrm{MIN}(S2)$.

6. Dealing with Periods

TSQL2's basic time element consists of a set of periods. For a drug, therefore, the duration of each prescription period is added up when computing the length of a prescription. To deal with individual prescription periods, TSQL2 introduces the special keyword PERIOD. Thus, to find drugs prescribed for more than 240 *consecutive* days, we have the following query:

Example 17 *TSQL2 Query: find the patients who have been prescribed some drugs for more than 240 consecutive days.*

```
SELECT SNAPSHOT Name
FROM Prescript(Name, Drug) (PERIOD) AS P
WHERE CAST(VALID(P) AS INTERVAL DAY)
        > INTERVAL '240' DAY
```

This solution suffers from several problems including the fact that (i) partitioning violates the TSQL2's data model [11] and (ii) we do not know how to extend this construct to query languages where there is no FROM clause.

Again, TSQL2's problem can be solved using a new aggregate called period which basically enumerates the periods in ascending temporal order. Time-points that fall within the same *consecutive* period of time are given the same period number (PerNo), and a different number is used for each period.

We can now define the following view:

Example 18 *A View Enumerating Periods*

```
CREATE VIEW PartitionedP(Name, Drug,
        PerNo, VTime)
AS SELECT P1.Name, P1.Drug,
        PERIOD(P2.VTime), P1.VTime
FROM Prescript AS P1 P2
WHERE P1.Name = P2.Name
   AND P1.Drug = P2.Drug
GROUP BY P1.Name, P1.Drug, P1.VTime
```

Now, PerNo can be used as one of the group-by attributes:

Example 19 *SQL^T Query for Example 17*

```
SELECT Name
FROM PartitionedP
GROUP BY Name, Drug, PerNo
HAVING LENGTH(VTime) > 240
```

An important advantage of this approach is that SQL^T can be defined completely using SQL-92. To compute PerNo for any time-point in an interval we must count the number of start-points of periods before it. Thus, the view PartitionedP could also have been defined as follows:

Example 20 *The meaning of* **PERIOD** *in SQL-92*

```
CREATE VIEW PartitionedP(Name, Drug,
      PerNo, VTime)
AS SELECT P1.Name, P1.Drug,
      COUNT(P2.VTime), P1.VTime
FROM Prescript AS P1 P2
WHERE P1.Name = P2.Name
   AND P1.Drug = P2.Drug
   AND P1.VTime >= P2.VTime
   AND NOT EXIST
      (SELECT P3.*
      FROM Prescript AS P3
      WHERE P3.VTime = P2.VTime - 1
         AND P3.Name = P2.Name
         AND P3.Drug = P2.Drug)
GROUP BY P1.Name, P1.Drug, P1.VTime
```

This definition is primarily of theoretical interest. The direct implementation of this aggregate is much more efficient—actually very efficient as we assume that the intervals are stored in ascending temporal order.

The previously created view can be useful for other queries as well. For instance, a query to find drugs whose first period of prescription to a patient was totally contained in 1996, is simply expressed as follows:

Example 21 *SQL^T Query: find drugs whose first prescription period is contained in 1996*

```
SELECT P.Drug
FROM PartitionedP AS P
WHERE P.PerNo = 1 AND
      ((SELECT C.VTime
      FROM Calendar AS C
      WHERE C.Year = 1996)
      CONTAINS
      (SELECT P1.VTime
      FROM Prescript AS P1
      WHERE P1.Name = P.Name
      AND P1.Drug = P.Drug))
GROUP BY P.Drug
```

This query is hard to express in TSQL2 but it is easy to be expressed in QBE^T and Datalog^T extended with a period aggregate and a length aggregate.

Example 22 *QBE^T Query for Example 21*

Prescript	Name	...	Drug	...	VTime
	G._name		G._drug		PERIOD.(_vtime)

PartitionedP	Name	Drug	PerNo	VTime
	_name	_drug	_perno	_vtime

PartitionedP	Name	Drug	PerNo	VTime
		P.G._drug1	_perno1	_vtime1

Calendar	Year	Month	Day	VTime
	1996			_vtime2

Conditions
_perno1 = 1 AND CONTAINS(_vtime2, _vtime1)

Example 23 *Datalog^T Query for Example 21*

```
partitionedP(Name, Drug, period⟨VTime⟩) ←
      prescript(Name, _, Drug, _, _, VTime),
query5(Drug, contains⟨(VTime2, VTime1)⟩) ←
      partitionedP(_, Drug, 1, VTime1),
      calendar(1996, _, _, VTime2).
```

In Datalog^T, period⟨VTime⟩ must return the original argument VTime along with its period number PerNo. On the other hand, contains evaluates to either true or false and returns zero arguments. Therefore, only the Name, Drug values for which the aggregate contains⟨(VTime2, VTime1)⟩ evaluates to true are produced in the head of the rule. The flexibility of having zero, one, or several values returned is supported in Version 5.1 of $\mathcal{LDL}$++ [16, 18].

7. Implementation

In this section, we discuss two alternative implementations of our universal temporal languages. One implementation is based on storing temporal intervals, and the other on storing events. Therefore, queries expressed against the point-based conceptual model must be translated into equivalent queries against an interval-based representation or an event-based representation. We will base our description on an implementation built at UCLA, where Datalog^T is supported on top of the $\mathcal{LDL}$++ system using extended aggregates [18, 19].

7.1. Interval-based implementation

Mapping to an interval-based relation at the internal level solves the space efficiency problem associated with the point-based data model used at the conceptual level. Then, tuples in our internal relations are timestamped with two time instants: one indicates the start-point and the other the end-point of the interval.

In the interval-based implementation, the database schema of Example 1 is translated into the following $\mathcal{LDL}$++ schema:

$$\begin{array}{l}\texttt{database}(\{\texttt{prescript}(\texttt{Name}:\texttt{string},\\ \quad\texttt{Physician}:\texttt{string},\texttt{Drug}:\texttt{string},\\ \quad\texttt{Dosage\_mg}:\texttt{integer},\\ \quad\texttt{Frequency\_Minute}:\texttt{integer},\\ \quad\texttt{Interval}:(\texttt{VTime},\texttt{VTime}))\}).\end{array}$$

We will now describe the mapping of queries on a point-based view to their interval-based equivalent using the syntactic framework of Datalog.

7.1.1. Selections, Projections and Joins

Take our query from Example 2 "What drugs have been prescribed with Proventil":

$$\begin{array}{l}\texttt{query6}(\texttt{Name},\texttt{Drug},\texttt{VTime1}) \leftarrow\\ \quad\texttt{prescript}(\texttt{Name},\_,\text{`}\texttt{Proventil}\text{`},\_,\_,\texttt{VTime1}),\\ \quad\texttt{prescript}(\texttt{Name},\_,\texttt{Drug},\_,\_,\texttt{VTime2}),\\ \quad\texttt{VTime1} = \texttt{VTime2},\\ \quad\texttt{Drug} \sim= \text{`}\texttt{Proventil}\text{`}.\end{array}$$

The equality $\mathsf{VTime1} = \mathsf{VTime2}$ on the point-based model can be translated into an explicit condition $\mathsf{intersect}(\mathsf{Int1}, \mathsf{Int2}, \mathsf{Int})$; in the interval-based representation this predicate returns every non-null intersection of $\mathsf{Int1}$ and $\mathsf{Int2}$ as Int. Since some columns of the original relation were dropped, an interval coalescing step is then performed using the aggregate coales. Therefore, we obtain the following equivalent rule:

$$\begin{array}{l}\texttt{query7}(\texttt{Name},\texttt{Drug},\texttt{coales}\langle\texttt{Int}\rangle) \leftarrow\\ \quad\texttt{prescript}(\texttt{Name},\_,\text{`}\texttt{Proventil}\text{`},\_,\_,\texttt{Int1}),\\ \quad\texttt{prescript}(\texttt{Name},\_,\texttt{Drug},\_,\_,\texttt{Int2}),\\ \quad\texttt{intersect}(\texttt{Int1},\texttt{Int2},\texttt{Int}).\\ \quad\texttt{Drug} \sim= \text{`}\texttt{Proventil}\text{`}.\end{array}$$

Once the intervals in $\mathsf{prescript}$ are arranged in ascending temporal order, the resulting values of Int can also be generated in that order; then, the computation of the coalescing aggregate coales can be performed quite efficiently.

In summary, a new $\mathsf{intersect}$ predicate is required for each temporal join specified by the original rule. Moreover, coales aggregation is required on the temporal argument when various columns have been projected out from goals in the body of the rule. A simple analysis shows coalescing is not needed for those rules where (i) there is no temporal argument in the head of the rule, or (ii) all variables appearing in the body of the rule also appear in its head. (These represent sufficient conditions, and more general syntactic conditions can be derived.) For instance, the rule

$$\texttt{q}(\texttt{X},\texttt{Y},\texttt{VTime}) \leftarrow \quad \texttt{p1}(\texttt{X},5,\texttt{VTime}),\texttt{p2}(\texttt{X},\texttt{Y},\texttt{VTime})$$

Will be rewritten using an $\mathsf{intersect}$ predicate but no coales aggregate. Consider now the following two rules:

$$\begin{array}{ll}\texttt{q1}(\texttt{Y},\texttt{VTime}) \leftarrow & \texttt{p1}(\texttt{X},\texttt{Y},\texttt{VTime}),\texttt{X} > 75.\\ \texttt{q2}(\texttt{Y},\texttt{VTime}) \leftarrow & \texttt{p1}(75,\texttt{Y},\texttt{VTime}).\end{array}$$

The first rule requires coalescing (i.e., it will be transformed by adding the aggregate coales in the head), while the second does not (and will be left unchanged).

7.1.2. Temporal Aggregates and Allen's Operators

An interesting aspect of our approach is that no rule or query transformation is needed for temporal aggregates. Instead, it is sufficient to provide a direct implementation of these aggregates on the interval representation. For instance, the first rule in Example 13 is kept without any change as:

$$\begin{array}{l}\texttt{groupdays}(\texttt{Name},\texttt{Drug},\texttt{length}\langle\texttt{VTime}\rangle) \leftarrow\\ \quad\texttt{prescript}(\texttt{Name},\_,\texttt{Drug},\_,\_,\texttt{VTime}).\end{array}$$

The implementation of length just requires an additional step after coales: once a maximal interval is constructed, its length is added to the running sum. In a similar fashion, we can implement the period aggregate assuming that the intervals are sorted in ascending start-time order: once a maximal interval is determined, the running *count* (rather than sum) is incremented. Also, the $\mathsf{contains}$ operator can be implemented in linear time once intervals are sorted [19].

7.2. Event-Based Implementation

In the event-based representation, we store changes between states, rather than the intervals in which the states hold. Thus, the following four entries,

```
prescript('Melaine', 'Dr. Frank', 'Proventil',
        3mgs, 360, ins(21,11,1996)).
```

```
prescript('Melaine', 'Dr. Frank', 'Proventil',
          3mgs, 360, del(28,11,1996)).
prescript('Melaine', 'Dr. Frank', 'Proventil',
          6mgs, 360, ins(28,11,1996)).
prescript('Melaine', 'Dr. Frank', 'Proventil',
          6mgs, 360, del(31,12,1996)).
```

describe the fact that initially Melaine was taking, under Dr. Frank's prescription, Proventil with dosage 3mgs, and frequency 360 minutes, from November 21st, 1996 till November 27th; then, the dosage was increased to 6mgs on November 28th, and the situation continued with no further change till December, 30, 1996. Therefore, ins represents inserts and del represent deletes; for simplicity, updates (e.g., increase in dosage to 6mgs) are hereby represented by a pair of delete/insert sharing the same time.

7.2.1. Selection, Projection and Join

The query in Example 2 "What drugs have been prescribed with Proventil" can now be implemented as follows:

```
query8(Name, Drug, coales⟨E⟩) ←
          prescript(Name, _, 'Proventil', _, _, E1),
          prescript(Name, _, Drug, _, _, E2),
          newevent(E1, E2, E),
          Drug ~= 'Proventil'.
```

The predicate newevent(E1, E2, E) is true if the last E1 event before E2 is an insertion, and, in this case, E = E2 (i.e., a joined event E is generated with the same type and timestamp as E2). Symmetrically, newevent(E2, E1, E) is true if the last E2 event preceding E1 is an insertion; then, E = E1. These operations can be performed efficiently once the events are stored in temporal order.

The conditions under which the aggregate coales must be included in the head of the rule are the same as for the interval-based implementation. The implementation of coales is also similar to that in interval-based implementation: a pass is made through the tuples sorted in ascending temporal order, and a running count is computed on the insert and delete events applied to the tuples till the current time (the number of ins must always be $\geq$ to that of del). Then, we only keep ins when the running count changes from zero to one, and we only keep del when the count changes from one to zero; all the other events are eliminated.

7.2.2. Temporal Aggregates and Allen's Operators

Again, no rewriting of the original rules for aggregates is needed as we move from a point-based representation to an event-based one. Here too, if we assume that events are stored by ascending temporal order, the implementation of period and length can be piggyback on that of coales. For period, we only need to keep a running count of all inserts generate by coales and both the insert and delete events are marked with a period number (PerNo). For length, when coales generates a new delete, the duration of time between the last insert and this delete must be added to the sum of intervals detected so far.

The implementation of all Allen's interval operators is straightforward, except for the contains operator. The comparison of two given intervals, each denoted by their insert event and delete event, is simple; the problem is that we are dealing with set of intervals, and we must pair-up the insert event with the delete event to ensure that they are from the same interval rather than from two different ones. Thus we must use pairs of events that have the same period number.

In a nutshell, the mapping of queries on the point-based representation to the event-based representation is simply accomplished by using few and well-behaved operators [5]. Similar considerations hold for interval-based representations.

8. Conclusion

Space limitations prevent us from discussing the large body of interesting research work previously published on temporal databases—see [8] and [10] for surveys of the topic.

Whereas query languages proposed in the past rely on the introduction of new temporal constructs, we have taken a minimalist's approach, and showed that current query languages provide all the syntactic constructs and semantic notions needed to express temporal queries as powerful as those expressible in TSQL2. While the standard SQL aggregates, such as sum, count, min and max, would suffice in terms of expressive power, we have added temporal aggregates to boost users' convenience and implementation efficiency.

In this paper, we proved the power and generality of this approach by showing that it can express all valid-time TSQL2 queries, and it can be extended uniformly to SQL, QBE, and Datalog. As a first step toward efficient implementation, we have also provided simple compilation strategies to translate these queries to equivalent ones executing on either an interval-based representation or an event-based one.

A cornerstone to the simplicity and generality of this approach, is the use of a point-based representation, whose benefits were first explored by Toman [14, 15]. Here, we have improved on that work by introducing temporal aggregates that model Allen's temporal operators and TSQL2's partitioning by user-friendly constructs amenable to efficient implementation [19]. We have also shown the *universality* of the approach (i.e., its validity with different query languages) and suggested that implementations based on *stored events* might be preferable to those using intervals.

Acknowledgements

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

This research was supported by NSF grant IRI-9632272.

References

[1] J.F. Allen. Maintaining knowledge about temporal intervals. In *Communications of the ACM, Vol.26, No.11*, pages 832-843, 1983

[2] 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

[3] 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

[4] T. Burns, et al. Reference Model for DBMS Standardization, Database Architecture Framework Task Group (DAFTG) of the ANSI/X3/SPARC Database System Study Group. In *SIGMOD Record Vol.15, No.1*, pages 19-58, 1986

[5] C.X. Chen and C. Zaniolo. Universal Temporal Data Languages. In *Proceedings of the 6th International Workshop on Deductive Databases and Logic Programming*, 1998

[6] C.S. Jensen, et al. A Consensus Glossary of Temporal Database Concepts. In *SIGMOD Record, Vol.23, No.1*, pages 52-64, 1994

[7] 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

[8] G. Ozsoyoglu and R.T. Snodgrass. Temporal and Real-Time Databases: A Survey. In *IEEE Transactions on Knowledge and Data Engineering, Vol.7, No.4*, pages 513-532, 1995

[9] R. Ramakrishnan. *Database Management Systems*, WCB/McGraw-Hill, 1998

[10] R.T. Snodgrass, I. Ahn, G. Ariav, D. Batory, et al. A TSQL2 Tutorial. In *SIGMOD Record, Vol.23, No.3*, pages 27-33, 1994

[11] R.T. Snodgrass, et al. *The TSQL2 Temporal Query Language*, Kluwer, 1995

[12] A. Tansel, et al. *Temporal Databases: Theory, Design and Implementation*, Benjamin/Cumming, 1993

[13] 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

[14] 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

[15] D. Toman. Point-Based Temporal Extensions of SQL and their Efficient Implementation. In *Temporal Databases: Research and Practice (O. Etzion, et al., eds.)*, Springer-Verlag, pages 211-237, 1998

[16] C. Zaniolo. A Short Overview of $\mathcal{LDL}$++: A Second-Generation Deductive Database System. In *Computational Logic, Vol.3, No.1*, pages 87-93, 1996

[17] C. Zaniolo, S. Ceri, C. Faloutsos, R. Snodgrass, and R. Zicari. *Advanced Database Systems*, Morgan Kaufmann, 1997

[18] C. Zaniolo, et al. $\mathcal{LDL}$++ Version 5. *http://www.cs.ucla.edu/ldl*, 1998

[19] C. Zaniolo and H. Wang. Logic-Based User-Defined Aggregates for the Next Generation of Database Systems. In *The Logic Programming Paradigm: Current Trends and Future Directions, K.R. Apt, V. Marek, M. Truszczynski, and D.S.Warren (eds.)*, Springer Verlag, 1999