Published in: *Proceedings of the 13th International Symposium on Temporal Representation and Reasoning (TIME-06)*, Budapest, Hungary, June 2006. IEEE Computer Society Press.

How Would You Like to Aggregate Your Temporal Data?

Michael H. Böhlen   Johann Gamper
Faculty of Computer Science
Free University of Bozen-Bolzano, Italy
{boehlen, gamper}@inf.unibz.it

Christian S. Jensen
Department of Computer Science
Aalborg University, Denmark
csj@cs.aau.dk

Abstract

Real-world data management applications generally manage temporal data, i.e., they manage multiple states of time-varying data. Many contributions have been made by the research community for how to better model, store, and query temporal data. In particular, several dozen temporal data models and query languages have been proposed.

Motivated in part by the emergence of non-traditional data management applications and the increasing proliferation of temporal data, this paper puts focus on the aggregation of temporal data. In particular, it provides a general framework of temporal aggregation concepts, and it discusses the abilities of five approaches to the design of temporal query languages with respect to temporal aggregation. Rather than providing focused, polished results, the paper's aim is to explore the inherent support for temporal aggregation in an informal manner that may serve as a foundation for further exploration.

1 Introduction

Most applications of database technology are temporal in nature. Examples include financial applications such as accounting, and banking; a broad range of record-keeping applications such as personnel, medical-record, and inventory management; scheduling applications such as airline, train, and hotel reservations and project management; and scientific applications such as weather monitoring. Applications such as these rely on *temporal databases*, which record time-referenced data.

A database models and records information about a part of reality, termed either the *modeled reality* or the *mini-world*. Aspects of the mini-world are represented in the database using a variety of structures or database entities—in the relational model, tuples are used. We will generally use the term *fact* for the logical statements about the mini-world that are recorded in the database.

Different temporal aspects may be associated with the facts stored in the database [12, 29]. Most importantly, the *valid time* of a fact is the collected times—possibly spanning the past, present, and future—when the fact is true in the mini-world. Valid time thus is used when capturing the time-varying states of the mini-world. All facts have a valid time by definition. However, the valid time of a fact may not necessarily be recorded in the database, for any of a number of reasons. For example, the valid time may not be known, or recording it may not be relevant.

Next, the *transaction time* of a database entity is the time when the entity is current in the database. Like valid time, this is an important temporal aspect. Transaction time is the basis for supporting accountability and "traceability" requirements, which exist in many applications, e.g., financial and medical applications.

The valid and transaction time values of database entities are drawn from some appropriate time domain. There is no single answer to how to perceive time in reality and how to represent time in a database. For example, the time domain may or may not stretch infinitely into the past and future; and time may be perceived as discrete, dense, or continuous. In databases, a finite, discrete, and totally ordered time domain is typically assumed, e.g., in the SQL standards.

Temporal data management can be very difficult using conventional (non-temporal) data models and query languages [24, 30, 33]. These provide little built-in support for managing such data, thus unnecessarily complicating database application development and leading to ineffective and inefficient ad-hoc solutions that must be reinvented each time a new application is developed. As a result, data management is currently an overly involved and error-prone activity. Temporal database research [3, 9, 10, 12, 37, 40] has produced several dozen proposals for temporal data models [31, 38] and query languages [4, 21, 22, 34, 39] that aim to remedy this situation.

This paper focuses on an increasingly important area of temporal data management, namely aggregation (e.g., [2, 6, 14, 23, 36, 41]). Aggregation gains in prominence in step with the increasing proliferation of temporal data and diffusion of business intelligence applications. In aggrega-

tion, an argument relation is transformed into a summary result relation. This is traditionally done by first partitioning the argument relation into groups of tuples with identical values for one or more attributes, then applying an aggregate function, e.g., count, to each group in turn.

We advocate a framework that generalizes traditional aggregation and offers orthogonal support for two aspects of aggregation [1, 2]: a) the definition of result groups for which to report one or more aggregate values and b) the definition of aggregation groups, i.e., collections of argument tuples that are associated with the result groups and over which the aggregate functions are computed. When aggregating temporal data, the time intervals to be associated with result tuples can depend on the actual data and are not known in advance.

Taking its outset in this framework, the paper explores the support for temporal aggregation in existing SQL-based temporal query languages that are based on tuple-timestamped valid-time data models. In particular, the paper considers five approaches to temporal query languages. As a vehicle for exploring each approach and for illustrating aspects of its inherent support, or lack thereof, for temporal aggregation, the paper considers the formulation of four aggregation queries in an SQL-based temporal query language that is prototypical for the approach. The findings for each approach are highlighted as observations that may serve as a basis for further study. To cover as many concepts as possible, the paper omits formal detail and is kept relatively informal.

Section 2 introduces an example, which is used throughout the paper and motivates the need for extensions of the SQL language to support temporal aggregation. Section 3 covers the notions of point-based and interval-based temporal data models. Section 4 then describes the paper's framework of temporal aggregation concepts. Section 5 proceeds to cover the different query language approaches. Finally, Section 6 summarizes, and Section 7 offers an outlook for temporal aggregation research.

2 Motivating Example

As a vehicle for illustration throughout the paper, consider the employee database in Figure 1. Relation EMPL captures work contracts with employees, recording for each contract the name of the employee who holds the contract (N), an identifier for the contract (CID), the department to which the employee is assigned for the duration of the contract, the monthly salary for the contract period (S), and the valid time of the contract (T).

The upper part of Figure 1 graphically illustrates relation EMPL, which contains four tuples. The valid time periods of the tuples are indicated by horizontal lines. For example, the first tuple states that Joe has a contract with the database department; the contract ID is 140, and Joe's monthly salary is 1200.

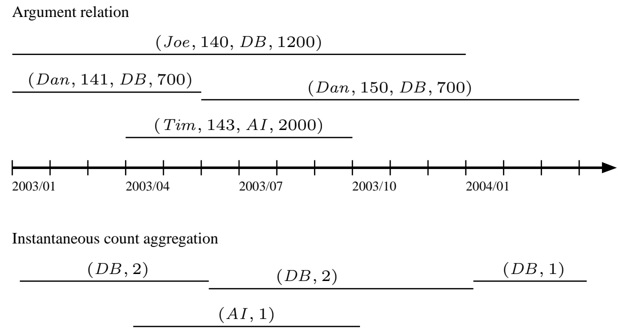

Figure 1. Temporal Aggregation

Given the prevalence of relational data management applications that manage time-varying data, one might question the need for a temporal query language. Is the existence of these applications not proof that SQL is sufficient for writing such applications? Put briefly, the reality is that in conventional query languages like SQL, temporal aggregation queries *can* be expressed, but in many cases only with great difficulty.

To illustrate the issue, consider the following aggregation query over relation EMPL that expresses the total number of contracts:

```
select count(*) as Cnt,
       [min(Ts),max(Te)] as T
from Empl
```

Here, `Ts` and `Te` denote the start and end point of a timestamp T, respectively, and `min` and `max` are well-known SQL aggregate functions. As we are in a temporal context, we choose to return a temporal relation, even if this query might also be interpreted as a non-temporal query. Hence, we assign the interval that lasts from the earliest start point to the latest end point of any argument tuple to the result.

The temporal generalization of this query, asking now for the time-varying count of contracts, as recorded in relation EMPL, is non-trivial to formulate. The intended result is shown in the lower part of Figure 1. Although possible, expressing this query in SQL is difficult. Although the two first tuples of the DB department have the same values for the non-temporal attributes, we do not combine them into one, since they have different lineage: different sets of contracts are responsible for these two result tuples. Reporting these two tuples instead of one yields a more informative result.

The point is that conceptually quite reasonable queries on temporal relations can be difficult to express using a query language such as SQL. Even SQL experts would be hard pressed to express the example temporal query in SQL.

Given also the ubiquitous nature of temporal data, this indicates a strong need for temporal support beyond what SQL offers today.

We proceed to explore in more detail the meaning of the association of a time interval with a tuple.

3 Point-Based and Interval-Based Models

The data model underlying a query language specifies the data structures that the query language manipulates. The numerous proposals for data models may be characterized according to a variety of criteria.

Within our scope of tuple-timestamped data models that capture valid time, we proceed to describe two types of data models, namely point-based and interval-based models. The former type of model inherently associates facts with time points, while the latter inherently associates facts with intervals.

3.1 Point-Based Temporal Data Models

The perhaps most basic type of temporal data model is capable of associating a fact with a set of time instants, or *points*.

This association may be achieved by timestamping each tuple with a single time point. Thus, if a fact is valid at several points in time, several so-called value-equivalent tuples (tuples that only differ in their timestamp) are used for capturing it, one for each time point.

As an illustration, part of the EMPL relation together with the result of the instantaneous temporal aggregation query [16, 23] from the previous section is shown in Figure 2, where timestamp attribute T stores valid time points at the granularity of months.

With point timestamping, syntactically different relations have different information content. Next, timestamps are atomic values that are easy to compare and manipulate. Assuming a totally ordered time domain, a standard set of comparison predicates, e.g., $=$, $\neq$, $<$, $>$, $\leq$, and $\geq$, is sufficient to conveniently compare timestamps.

Point timestamping is often considered only as a basis for the design of query languages and is not meant for physical representation. Indeed, for all but the most trivial time domains and facts, the space needed when using the point model is prohibitive. Point timestamps are also rarely a user-friendly format for the display of temporal relations.

Due to their simplicity, point timestamped temporal data models have been popular in theoretical studies, including constraint databases (cf., e.g., [7, 13, 25, 26, 27]).

Another approach to associating facts with time points is to timestamp tuples with *sets of time points* or with so-called *temporal elements*, which are finite unions of time intervals. With these representations, a relation does not contain value-equivalent tuples—all value-equivalent tuples in the corresponding point timestamped relation are combined into one tuple, with a timestamp that captures all the time points of those tuples.

N	*CID*	*D*	*S*	*T*
Joe	140	*DB*	1200	2003/01
		. . .		
Joe	140	*DB*	1200	2003/12
Dan	141	*DB*	700	2003/01
Dan	141	*DB*	700	2003/02
Dan	141	*DB*	700	2003/03
Dan	141	*DB*	700	2003/04
Dan	141	*DB*	700	2003/05
Dan	150	*DB*	700	2003/06
Dan	150	*DB*	700	2003/07
Dan	150	*DB*	700	2003/08
Dan	150	*DB*	700	2003/09
Dan	150	*DB*	700	2003/10
Dan	150	*DB*	700	2003/11
Dan	150	*DB*	700	2003/12
Dan	150	*DB*	700	2004/01
Dan	150	*DB*	700	2004/02
Dan	150	*DB*	700	2004/03
Tim	153	*AI*	1800	2003/04
		. . .		
Tim	153	*AI*	1800	2003/09

(a) Relation EMPL

D	*Cnt*	*T*
DB	2	2003/01
DB	2	2003/02
DB	2	2003/03
DB	2	2003/04
DB	2	2003/05
DB	2	2003/06
DB	2	2003/07
DB	2	2003/08
DB	2	2003/09
DB	2	2003/10
DB	2	2003/11
DB	2	2003/12
DB	1	2004/01
DB	1	2004/02
DB	1	2004/02
AI	1	2003/04
AI	1	2003/05
AI	1	2003/06
AI	1	2003/07
AI	1	2003/08
AI	1	2003/09

(b) Instantaneous Aggregation

Figure 2. Point Timestamping

Yet another approach to associating facts with time points is to timestamp tuples with *intervals*. Multiple tuples are then needed if a fact is valid over a non-convex set of time points.

Figure 3(a) illustrates the approach, where the timestamp attribute $T = [T_s, T_e]$ stores a valid time interval represented by its inclusive start and end point, respectively.

When employing intervals rather than time points as timestamps, two timestamps satisfy precisely one of the following thirteen relationships, first enumerated by James Allen: *before*, *meets*, *overlaps*, *during*, *starts*, *finishes*, and *equal*, in addition to the inverses of the first six of these. Allen's pioneering work in this area has inspired designs of many of the collections of interval predicates available in temporal query languages. While well-chosen interval predicates are more convenient to use than relationships over interval start and end points, such predicates alone turn out to not be sufficient to provide comprehensive and easy-to-use support for temporal data management in general and the point-based view in particular.

As we will see in Section 5, using interval timestamps as compact representations of sets of time points has the effect of making some conceptually simple queries cumbersome

to formulate.

The notion of *snapshot equivalence*, which reflects a point-based view of data, establishes a correspondence between interval timestamped relations. Consider the two instances of the result relation in Figure 3. The relations are different, but snapshot equivalent, meaning that they contain the same snapshots. Specifically, the relation in Figure 3(c) is a coalesced version of the relation in Figure 3(b). In *coalescing*, value-equivalent tuples with adjacent or overlapping time intervals are merged.

N	CID	D	S	T
Joe	140	DB	1200	[2003/01,2003/12]
Dan	141	DB	700	[2003/01,2003/05]
Dan	150	DB	700	[2003/06,2004/03]
Tim	143	AI	2000	[2003/04,2003/10]

(a) Relation EMPL

D	Cnt	T
DB	2	[2003/01,2003/05]
DB	2	[2003/06,2003/12]
DB	1	[2004/01,2004/03]
AI	1	[2004/04,2004/09]

(b) Result Relation: Instance 1

D	Cnt	T
DB	2	[2003/01,2003/12]
DB	1	[2004/01,2004/03]
AI	1	[2004/04,2004/09]

(c) Result Relation: Instance 2

Figure 3. Interval Timestamping

3.2 Interval-Based Temporal Data Models

Interval-based temporal data models associate facts with intervals. In such models, intervals are not just compact representations of time points.

While several different types of timestamps may be used in point-based temporal data models, it is most natural to use interval timestamps for interval-based models (although timestamps that are sets of intervals could also be considered).

To illustrate the difference between point- and interval-based models, recall the query result displayed in the lower part of Figure 1. This result contains more information than what is given in the point-timestamped result displayed in Figure 2(b). Put differently, this result cannot be reconstructed from the result in Figure 2(b). (An additional attribute, such as *CID* in relation EMPL, may perhaps be used for this purpose. The next section will return to this issue.)

As yet another manifestation of the extra information captured by interval-based models, snapshot equivalent relations may have different information content in such models, as is the case in Figures 3(b) and 3(c). The two relations are different, also from a semantic point of view. Further, it is not appropriate to require relations in interval-based models to be coalesced. This would imply that it is not possible to distinguish between two consecutive contracts with the same values, unless an additional attribute such as a contract identifier is introduced. For example, in Figure 3(a), attribute *CID* allows us to separate tuples two and three, for which all other non-timestamp attributes have the same value.

So in the interval-based models that we will use as our outset, intervals are not merely representational devices—they carry meaning beyond denoting sets of points. Returning to the example, two consecutive contracts with the same values are clearly different from a single contract over the whole period. They require no additional attribute to identify the contracts, and the instance in Figure 3(b) is the appropriate representation of our result relation.

4 Temporal Aggregation

We proceed to describe a general framework for temporal aggregation and then consider four examples of temporal aggregation.

4.1 Temporal Aggregation Framework

As an outset for the temporal aggregation framework, recall that Klug's (and SQL's) conventional framework for non-temporal aggregation performs aggregation on an argument relation according to two parameters [17]:

1. a set of attributes drawn from the argument relation, termed grouping attributes; and
2. a set of pairs of a new attribute name and an aggregation function.

The tuples in the argument relation are partitioned according to their values for the grouping attributes. Then for each partition, each aggregate function given in the second parameter is computed on the tuples in the partition, and the result is stored as a value of the associated attribute for each tuple in the partition. Finally, the non-grouping attributes of the argument relation may be eliminated from the result by means of a projection using relational algebra.

We propose a temporal aggregation framework that generalizes the non-temporal one in two important respects. Instead of partitioning the tuples in the argument relation according to their values for certain of their attributes, we introduce a separate *grouping table* that contains a tuple for each group to be represented in the query result. This table generally has as attributes a subset of the attributes of the input relation, the timestamp attribute being one of them. Additional, new attributes may also be included.

Second, we introduce a parameter that maps tuples from the input relation to tuples in the grouping table. This *mapping function* may assign the same argument tuple to zero, one, or many groups. This differs from the conventional

framework, where each input tuple is mapped to exactly one group.

The new framework retains the second parameter from the conventional framework.

The resulting framework generalizes the specification of result groups, it generalizes the mapping of input tuples to result groups, and it decouples the specification of result groups from the mapping of input tuples to the result groups.

An important aspect of the framework is that the values for the timestamp attribute in the tuples in the grouping relation may be either fixed or inferred from the data in the input relation. The case of fixed intervals corresponds to how the non-timestamp attribute values are treated: they must be provided explicitly.

The case of inferred intervals is specific to the timestamp attribute. An inferred interval is calculated as the intersection of the intervals associated with the argument tuples that contribute to the aggregate results to be associated with the group, or grouping tuple, that the inferred intervals applies to. These inferred intervals are termed *constant* because there are no changes in the argument relation during these intervals. Constant intervals are non-overlapping and maximal.

4.2 Example Queries

The following queries together with their intended results build on the employee database. They serve to illustrate the concepts in the aggregation framework and will also be used for illustration in the rest of the paper. The result relations of the queries are illustrated in graphical form in Figure 4.

Query Q_{ci} (constant intervals): *For each department, what is the time-varying number of contracts?*

D	Cnt	T
DB	2	[2003/01,2003/05]
DB	2	[2003/06,2003/12]
DB	1	[2004/01,2004/03]
AI	1	[2003/04,2003/09]

Query Q_{ci} is an example of an instantaneous aggregation [16, 23] that must be applied to each database state. To compute the result at a specific time point, all tuples that are valid at that time point are considered.

The attributes of the grouping table for this query are the department attribute D and timestamp attribute T. The table has two tuples, namely one with DB and one with AI as its D value. These tuples have unspecified timestamps, as these are inferred as the constant intervals from the argument relation.

Thus, for each group, the result contains a tuple for each aggregate value and each constant interval associated with that value. For example, the group for department DB has two aggregate values (1 and 2), and aggregate value 2 holds for two constant intervals.

Query Q_{fi} (fixed intervals): *For each department, how many contracts were in effect during each half-year?*

D	Cnt	T
DB	3	[2003/01,2003/06]
DB	2	[2003/07,2003/12]
DB	1	[2004/01,2004/06]
AI	1	[2003/01,2003/06]
AI	1	[2003/07,2003/12]
AI	0	[2004/01,2004/06]

Query Q_{fi} has the same non-temporal part as Q_{ci}, but here the query explicitly specifies *fixed time intervals* [2] over which to evaluate the non-temporal aggregation.

As in the previous example, the grouping table has the department and the timestamp as its attributes. However, now the timestamp attribute values are specified explicitly. For each department, there are three half-year intervals during which contracts are in effect. The grouping table therefore contains six tuples.

For each of the resulting six groups, a count of contracts is computed by considering all contracts that match the department value for the group and have a timestamp that overlaps the six-month period of the group.

Query Q_{cum} (cumulative aggregation): *At each time, what is the number of contracts within the last three months?*

Cnt	T
2	[2003/01,2003/03]
3	[2003/04,2003/05]
4	[2003/06,2003/07]
3	[2003/08,2003/11]
2	[2003/12,2004/02]
1	[2004/03,2004/05]

Query Q_{cum} is a *cumulative aggregation* query, also termed a moving-window query [41, 42]. It slides along the time line, computing at each time point an aggregate that takes into consideration all tuples that were valid at some point during the past three month. In general, the value of a cumulative aggregate at time point t is computed over all tuples whose valid intervals overlap with the interval $[t-w, t]$, where w is the window offset.

In the result relation, tuples over consecutive time points that have the same aggregate value and identical lineage information are coalesced. In this query, the grouping table

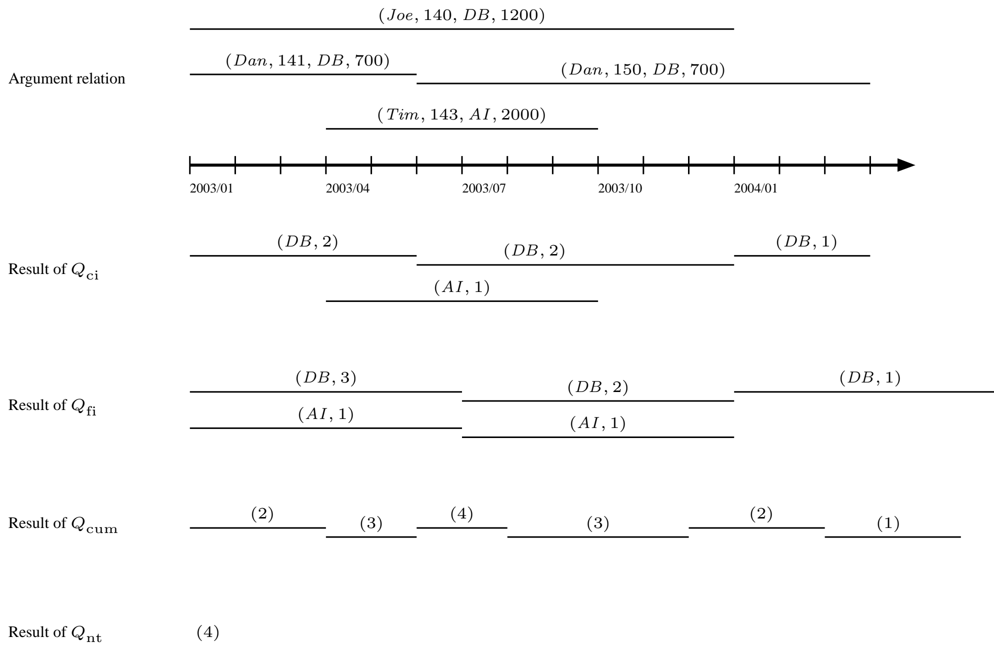

Figure 4. Temporal Aggregation Results

has only one attribute, the timestamp, which is inferred from the argument tuples. The argument tuples associated with a group are all those tuples that were valid within the past three months.

Query Q_{nt} (non-temporal aggregation): *What is the number of contracts in total?*

Cnt
4

In Query Q_{nt}, the aggregation is to be applied to the entire relation independently of any temporal information, producing one result tuple that contains the total number of contracts in the database. Thus, the grouping table is empty.

In the following, we discuss the support for temporal aggregation inherent in different approaches to temporal query language design, using these four queries as examples.

5 Analysis of Temporal Query Languages

This section discusses the support for temporal aggregation inherent in five distinct approaches to temporal query language design. To be specific, the section bases its discussion of each approach on a specific temporal extension to the SQL query language that is prototypical to the approach.

Because the focus is on the inherent properties of the approaches, we gloss over semantic variations among temporal constants and predicates (e.g., overlaps), and we introduce additional functions (e.g., duration) into the prototypical languages as needed. We even permit ourselves to be liberal with respect to available language constructs, to the extent that this is helpful in better representing the approaches.

We proceed to first discuss building blocks that will prove helpful in formulating temporal aggregation queries in several of the approaches covered. Then each approach is covered in turn.

5.1 General Building Blocks

There are a few concepts that are fundamental when expressing temporal aggregation queries and that are either not supported or barely supported in current temporal query languages. These concepts concern the computation of the timestamps for the result tuples that depend on the data in the argument relation and possibly also on the query. We present these concepts next and use them in the subsequent analysis of query languages.

Computation of Constant Intervals. Query Q_{ci} requires the computation of constant intervals, i.e., the intervals over which the sets of argument tuples do not change. Expressing these intervals in SQL is possible, but unreasonably complicated, as illustrated by the following solution that

uses two views.

```
create view EndPoints (D,TP) as
  select distinct D, Ts as TP from Empl
  union
  select distinct D, Te as TP from Empl

create view CI (D,T) as
  select a.D, [a.TP,b.TP] as T
  from EndPoints as a, EndPoints as b
  where a.D = b.D
    and a.TP < b.TP
    and not exists(
              select *
              from EndPoints as c
              where a.TP < c.TP < b.TP)
    and exists(
          select *
          from Empl as d
          where overlaps(d.T,[a.TP,b.TP]))
```

The view `EndPoints(D,TP)` is defined by the argument relation and determines all distinct start and end points of the argument tuples grouped by department. These time points are the end points of the constant intervals. The view `CI(D,T)` is defined over these end points and extracts those combinations of end points that form the valid constant intervals over which the result tuples are defined. Two end points t and t' of the argument relation form a constant interval $[t, t']$ if there are no end points in-between and there is an argument tuple that overlaps with the time interval $[t, t']$.

Note that the computation of the constant intervals needs to take into consideration the non-temporal grouping attributes, and hence, the above SQL statement depends on the query. Moreover, the expression is not only syntactically complicated, but also expensive to compute.

Chron Relation. An abstract unary `Chron` relation has been proposed that has a single temporal attribute that stores all possible chronons (time points) of the temporal universe [39]. Such a `Chron` relation is helpful when expressing a broad range of queries. For the use of this relation to be practical, implementation level solutions have to be developed that do not require the materialization of the `Chron` relation.

Consider query Q_{fi}, which explicitly involves the periods during which a result tuple is expected, i.e., every semester where a tuple is valid. Using the `Chron` relation, we can construct these semesters as follows:

```
create view FI (D,T) as
  select distinct D, sem(a.T) as T
  from Chron as a, Empl as b
  where overlaps(sem(a.T),b.T)
```

The function `sem(T)` takes as argument a chronon and returns the semester to which this chronon belongs. For example, $sem(2003/01)$ returns the first semester in 2003 represented as an interval: $[2003/01, 2003/06]$.

Again, note that the non-temporal grouping attributes of the specific query have to be considered, which precludes a general solution for all queries with fixed intervals.

Timestamp Generation. Another non-standard feature that facilitates the formulation of temporal statements is the availability of generative constructs: general functions that return sets of values that are then further processed. Examples include functions that generate all time points included in an interval or all semesters covered by an interval.

Being more precise, we assume a user-defined function f that takes as input a timestamp T and returns a set of intervals, i.e.,

$$f(T) = \{I_1, \ldots, I_m\}$$

This function can then be used in the query language and has the following semantics:

$$SQL(f(T)) \equiv SQL(I_1) \cup \cdots \cup SQL(I_m)$$

That is, we evaluate the SQL query for each of the intervals returned by f and take the union of the result tuples.

5.2 Approach I: Abstract Data Types

The earliest and, from a language design perspective, simplest approach to improving the temporal data management capabilities of a query language is to introduce time data types and associated predicates and functions.

Observation 1 *Adding a new ADT to SQL is attractive because it has limited impact on SQL and because the extension of SQL with new data types with accompanying predicates and functions is fairly well understood.*

Formulations of predicates on time-interval data types have been influenced by Allen's 13 interval relationships. With reference to these, different sets of practical proposals for predicates have been proposed. To illustrate this approach, we assume that the employee relation is represented by the interval-timestamped relation in Figure 3.

$Q_{\text{ci}}^{\text{SQL}}$**:** As mentioned in Section 2, expressing a time-varying aggregation as in Q_{ci} is possible, but there exists no reasonable SQL solution. Using the views discussed above, we can express Q_{ci} as follows:

```
select a.D, count(*) as Cnt, a.T
from CI as a, Empl as b
where a.D = b.D and overlaps(a.T,b.T)
group by a.D, a.T
```

It is evident from this example that the computation of the constant intervals is the hard part, while the computation of the aggregate function is quite straightforward and needs just a Boolean function to test the overlapping of timestamps.

Observation 2 *Instantaneous temporal queries are complex to formulate with standard SQL extended with an interval-based ADT.*

$Q^{\mathrm{SQL}}_{\mathrm{fi}}$**:** Query Q_{fi} explicitly specifies the periods for which a result tuple is expected, i.e., for every semester where data are present. We use the `Chron` relation and the `sem` function introduced above and express the query as follows:

```
select b.D, count(*) as Cnt, a.T
from FI as a, Empl as b
where a.D = b.D and overlaps(a.T,b.T)
group by b.D, a.T
```

The only difference to Q_{ci} is in the computation of the timestamps of the result tuples.

$Q^{\mathrm{SQL}}_{\mathrm{cum}}$**:** Query Q_{cum} is similar to Query Q_{ci} in that the aggregate function is computed for each time point, and consecutive time points with the same result value and identical lineage are coalesced. The timestamps of the result tuples can be computed from the argument tuples similar to how it was done for the constant intervals. However, the length of the moving window has to be considered. The timestamps of the result tuples extend beyond the timestamps of the argument tuples. The following view computes the possible end points of the result tuples.

```
create view EndPoints (TP) as
  select distinct Ts as TP from Empl
  union
  select distinct Te+2 as TP from Empl
```

We must extend all end points of the argument tuples by the value 2. Based on the end points, a view `CumI` can be defined that is identical to `CI` for constant intervals, except that for this query, there are no non-temporal grouping attributes.

With the view `CumI` in place, we can formulate Query Q_{cum} as follows:

```
select count(*) as Cnt, a.T
from CumI as a, Empl as b
where overlaps([a.Ts-2,a.Te],b.T)
group by a.T
```

As for the computation of the timestamps, we have to consider again the length of the moving window and to aggregate over all argument tuples that overlap an interval that starts two chronons before the timestamp of the result tuple.

$Q^{\mathrm{SQL}}_{\mathrm{nt}}$**:** Counting the total number of contracts in the database is straightforward:

```
select count(*) as Cnt
from Empl
```

In summary, the availability of appropriate time data types aids only little in the formulation of temporal aggregation queries. We identify two core problems that make temporal queries complex. First, the calculation of the timestamps for queries with constant intervals as well as cumulative aggregates is complex. Second, the calculation of the timestamps for Query Q_{fi} refers to the `Chron` relation. This relation cannot be materialized.

5.3 Approach II: Fold/Unfold

Being of fixed size, interval timestamps are very convenient when capturing the temporal aspects of information. In some respects, the most straightforward and simplest means of capturing temporal aspects is to include an extra interval-valued time attribute in each relation. However, one might also suspect that the difficulty in formulating temporal queries in the previous section is caused by the intervals. SQL comes unprepared to support something (an interval) that represent something (a set of consecutive time points) that it is not.

In response to this, it has been proposed to equip SQL with the ability to *normalize* timestamps. The idea is to split or merge interval timestamps so that they are *aligned* (identical or disjoint) and can be treated as atomic entities.

Advanced most prominently by Lorentzos and his colleagues [18, 19, 20, 21], the earliest and most radical approach is to introduce the two functions *unfold* and *fold*. The unfold function decomposes an interval timestamped tuple into a set of point timestamped tuples, one for each point in the original interval. The fold function "collapses" a set of point timestamped tuples into value-equivalent tuples timestamped with maximum intervals.

Observation 3 *Extending SQL with functions fold and unfold is attractive because of its conceptual simplicity.*

The idea is to use the interval-based representation of temporal information while being able to manipulate it as if the point-based representation was used, thus obtaining the representational benefits of intervals while avoiding the problems they seem to pose in query formulation.

The general pattern for queries using unfold and fold is to:

1. explicitly construct the point-based representation by unfolding the argument relation(s);

2. compute the query on interval-free representation; and

3. fold the result to end up with an interval-based representation.

Observation 4 *Transitioning from the interval to the point representation puts a load on the database system that is exponential in the length of the intervals.*

The fold and unfold functions have been integrated into IXSQL [18, 21], which we use for illustration. IXSQL inherits and extends the semantics of SQL. Thus, each SQL query is also an IXSQL query. In the discussion below we assume the EMPL relation in Figure 3.

$Q_{\text{ci}}^{\text{IXSQL}}$**:** Query Q_{ci} that expresses the time-varying number of contracts per department can be formulated as follows:

```
select D, count(*) as Cnt, T
from ( select *
       from Empl
       reformat as unfold T )
group by D, T
reformat as fold T
```

The inner query unfolds the argument relation yielding the point-based representation shown in Figure 2(a). Then the aggregation is computed on this relation and with the `fold` function transformed back into a interval-stamped relation. Note that the obtained result is different from the intended result in Figure 3(b). The normalization step does not carry over any lineage information, and the unfold operation creates maximal intervals of snapshot equivalent tuples independently of the argument tuples that produce the result. In particular, the first two intended result tuples are merged into a single tuple, and we get the result shown in Figure 3(c).

Observation 5 *When transitioning from intervals to points any semantics associated with the intervals is lost.*

$Q_{\text{fi}}^{\text{IXSQL}}$**:** To express Query Q_{fi}, we unfold the argument relation and determine all semesters for which data are available:

```
select D, count(*) as Cnt, S as T
from ( select distinct D, sem(T) as S
       from Empl
       reformat as unfold T ) as a,
     Empl as b
where a.D = b.D and a.S cp b.T
group by D, a.S
reformat as fold T
```

Again, the `unfold` function first transforms the interval-timestamped relation into a point-timestamped relation, from which the different pairs of departments and semesters are extracted. The IXSQL predicate `cp` corresponds to the overlaps function and tests for common time points of the two arguments.

$Q_{\text{cum}}^{\text{IXSQL}}$**:** The cumulative aggregation query follows the pattern of the previous two queries: we unfold the EMPL relation so we can work with time points, and use a join to match it with tuples within the specified window.

```
select count(*) as Cnt, a.T
from ( select *
       from Empl
       reformat as unfold T ) as a,
     ( select *
       from Empl
       reformat as unfold T ) as b
where b.T >= a.T-2 and b.T <= a.T
group by T
reformat as fold T
```

Note that the two time points after the very last argument tuple (cf. Figure 4) are missing. This can be fixed by extending the inner SQL statement with a union statement that explicitly adds these points.

As in the case with constant intervals, the transformation into interval-timestamped result tuples by the `fold` operation yields the coalesced relation in Figure 3(c), which is not the intended result.

$Q_{\text{nt}}^{\text{IXSQL}}$**:** The standard SQL solution can be used to count the total number of contracts:

```
select count(*) as Cnt
from Empl
```

In summary, a language enriched with folding and unfolding offers some support for expressing instantaneous aggregation with constant intervals as in Query Q_{ci}. However, the final fold function, coalescing snapshot equivalent tuples of the point model into tuples over maximal intervals, might lead to wrong results.

Regarding fixed intervals, IXSQL provide no generic support. Although the language offers a window function to generate windows of a specific size with a determined offset, it is not expressive enough to formulate sliding windows or an arbitrary number of consecutive timestamps.

The efficient evaluation of queries formulated using fold and unfold has yet to be resolved. Unfolding has a worst case space complexity that is exponential (an m bit binary integer encodes up to $2^m - 1$ database states); and for the time domains available in current systems, unfolded relations are so large that storing them is impractical.

A more subtle observation is that IXSQL adopts a view on relation instances that is neither purely point-based nor interval-based. It is not purely point-based because it is sensitive to the specific interval representation chosen for the

data. Thus, when different, but snapshot-equivalent, relations are used, the same query generally returns different results. In contrast, the fold and unfold functions only preserve the information content in a relation up to that captured by the point-based view. For example, unfolding and then folding the relation instance in Figure 3(b) yields the instance in Figure 3(c).

Finally, it may be noted that the three-step procedure for using fold and unfold is exactly a procedure and thus adds a slight procedural element to SQL, the core of which may be seen as being declarative.

5.4 Approach III: Point Timestamps

A more radical approach to designing a temporal query language is to simply assume that temporal relations use point timestamps—fold and unfold are then not needed. The temporal query language SQL/TP advanced by Toman takes this approach to generalizing queries on non-temporal relations to apply to temporal relations [5, 39]. The semantics of SQL/TP is defined with respect to the point-based representation, and we thus assume the EMPL relation instance in Figure 2 in the following. The restriction to point timestamps yields a simple and unambiguous semantics that avoids many of the pitfalls that can be attributed to interval timestamps.

Observation 6 *SQL/TP does not permit the association of information with intervals.*

The strength of SQL/TP is in its generalization of queries on snapshot relations to corresponding queries on corresponding temporal relations. The general principle is to extend the snapshot query with equality constraints on the timestamp attribute of the temporal relation, to separate different database snapshots during query evaluation.

$Q_{\mathrm{ci}}^{\mathrm{SQL/TP}}$**:** Query Q_{ci} is straightforward to express in SQL/TP, as the argument tuples are first grouped by department and time points, upon which the aggregate function is computed.

```
select D, count(*) as Cnt, T
from Empl
group by D, T
```

The grouping takes care of isolating the database states from one another. This query is restricted to finite (discrete and bounded) time domains, to avoid infinite relations and counts.

Observation 7 *The semantics of SQL/TP statements is defined with respect to the point representation, which is different from the presentation of a temporal relation.*

$Q_{\mathrm{fi}}^{\mathrm{SQL/TP}}$**:** The computation of Q_{fi} is more complicated. We have to group the time points into semesters, and each time point of a semester must produce the same aggregate value.

```
select a.D, count(*) as Cnt, a.T
from Empl as a, Empl as b
where a.D = b.D
and a.T-1 div 6 = b.T-1 div 6
group by a.D, a.T
```

The condition in the where clause groups the argument tuples by department and semester. The aggregate function is computed over these groups and assigned to each time point in the semester. For example, the result of the first semester in 2003 is as follows:

D	*Cnt*	*T*
DB	3	2003/01
DB	3	2003/02
DB	3	2003/03
DB	3	2003/04
DB	3	2003/05
DB	3	2003/06
AI	1	2003/01
AI	1	2003/02
AI	1	2003/03
AI	1	2003/04
AI	1	2003/05
AI	1	2003/06

$Q_{\mathrm{cum}}^{\mathrm{SQL/TP}}$**:** SQL/TP does not provide any natural support for the formulation of cumulative queries, i.e., a mechanism to move a window of fixed size over the time line and to produce a result at each time point. Hence, Query Q_{cum} is more complex:

```
select count(distinct CID) as Cnt, a.T
from ( select distinct T
       from Empl ) as a,
     Empl as b
where a.T-2 <= b.T < a.T
group by a.T
union
select count(distinct CID), max(a.T)+1
from ( select distinct T
       from Empl ) as a,
     Empl as b
where max(a.T)-1 <= b.T < max(a.T)+1
group by a.T
union
select count(distinct CID), max(a.T)+2
from ( select distinct T
       from Empl ) as a,
     Empl as b
where max(a.T) <= b.T < max(a.T)+2
group by a.T
```

Note that the query is a union of three almost identical parts. The last two parts take care of the two very last time points (cf. Figure 4) that are not part of the timestamps of the original relation.

$Q_{\mathrm{nt}}^{\mathrm{SQL/TP}}$**:** Since SQL/TP counts the number of tuples in the abstract relation, it is necessary to project the time attribute and eliminate duplicates. This yields the intended result if the tuples are distinguishable. In our case, the contract ID ensures this.

```
select count(distinct CID) as Cnt
from Empl
```

This query again requires the use of a contract identifier in order to be able to distinguish between different contracts. The timestamps alone do not provide any information about this.

Observation 8 *Aggregates in SQL/TP compute the aggregate with respect to the abstract temporal relation. Operations such as counting the numbers of rows in a concrete representation are not possible.*

In one sense, SQL/TP and SQL are opposites when it comes to the handling of temporal information. In SQL, intervals have no special meaning—they are treated as atomic entities. In contrast, SQL/TP effectively decomposes intervals into sets of points. This difference becomes clear when considering aggregate queries. In SQL, time-varying aggregation (Q_{ci}) is poorly supported, while SQL/TP needs to resort to auxiliary attributes for "time-invariant" aggregation (Q_{nt}).

In several of the examples, we have used relations with contract IDs in order to be able to capture the intended information and express the desired queries. While the reliance on contract identifiers appears to be a minor issue, it is worth noting that such identifiers do not offer a systematic approach to obtaining point-based semantics *and* a semantics that preserves the intervals of the argument relations.

The problem is that set operations as well as aggregation are sensitive to any additional attributes and essentially do not permit the presence of such attributes. This issue is not germane to SQL/TP, but seems to apply equally to any approach that uses a point-based data model.

In summary, the strength of SQL/TP is its restriction to time points that ensures a simple and well-defined semantics. As intervals are still to be used in the physical representation of the temporal information as well as when presenting the results of queries to the users, one may think of SQL/TP as a variant of IXSQL where, conceptually, queries must always apply unfold as the first operation and fold as the last. A compilation technique has been supplied for SQL/TP that avoids this unfolding, thus offering hope that SQL/TP queries can be evaluated efficiently in practice [].

5.5 Approach IV: Syntactic Defaults

Along with the introduction of temporal abstract data types, what may be termed *syntactic defaults* have been introduced that make the formulation of common temporal queries more convenient. The most common defaults concern access to the current state of a temporal database and for handling temporal generalizations of non-temporal queries, e.g., joins. The most comprehensive approach based on syntactic defaults is the TSQL2 language [32, 35], which we use for exemplification. We assume the EMPL instance in Figure 3.

In TSQL2, a default valid clause, placed after the select clause, computes the intersection of the valid times of the tuples in the argument relations mentioned in the from clause, which is then returned in the result. For example, the timestamp of a tuple that results from joining two relations is the intersection of the timestamps of the two argument tuples that produce the tuple. With only one relation in the from clause, this yields the original timestamps.

In order to compute an instantaneous temporal aggregation, the timestamps of overlapping argument tuples that belong to the same group must be intersected. This computation of constant intervals cannot be expressed easily in SQL (cf. Section 5.1). Moreover the interaction with the default valid clause described above is not clear to the authors. This is taken to be evidence of the complexity of a language that provides comprehensive syntactic defaults. It also implies that the queries described in this section may not be correct. We rely on the description of temporal aggregates by Kline et al. [15].

Observation 9 *Well-chosen syntactic defaults yield a language that allows to succinctly formulate common temporal queries.*

$Q_{\mathrm{ci}}^{\mathrm{TSQL2}}$**:** To formulate an instantaneous aggregation, it is possible to extend the group by clause with a valid clause. In the query below, the term `using instant` is in fact the default and could be omitted. We added it for clarity since `valid(Empl)` denotes the original timestamps and we want to group according to constant intervals, not the original timestamps.

```
select D, count(*) as Cnt
from Empl
group by D, valid(Empl) using instant
```

$Q_{\mathrm{fi}}^{\mathrm{TSQL2}}$**:** Grouping into periods is supported through the `using` clause. In this case, we specify a grouping of 6 months (i.e., one semester). In passing, we mention that we are uncertain whether this indeed yields January–June and July–December or whether shifted semesters might result.

```
select D, count(*) as Cnt
from Empl
group by D, valid(Empl) using 6 month
```

$Q_{\mathrm{cum}}^{\mathrm{TSQL2}}$**:** TSQL2 provides native support for cumulative (moving-window) aggregates. Specifically, the group by clause allows specification of a leading and trailing time interval for a moving window. Hence, Q_{cum} can be expressed as follows:

```
select count(*)
from Empl
group by D, valid(Empl) leading 2 month
```

$Q_{\mathrm{nt}}^{\mathrm{TSQL2}}$**:** The default behavior of TSQL2 is to return temporal relations. The `snapshot` keyword is used for retrieving non-temporal relations. Thus, to retrieve the total number of contracts, we can use the following non-temporal aggregation:

```
select snapshot count(*) as Cnt
from Empl
```

TSQL2 is a large language with many parts and an informally specified semantics. It provides syntactic defaults that serve as shorthands and thus simplify the formulation of temporal queries over point-based temporal databases. The problem with syntactic defaults relates to lack of the "scalability" over language constructs. When defining a language that uses syntactic defaults, one must explicitly specify a large number of defaults. When extending a large and non-orthogonal language such as SQL, it becomes challenging to be comprehensive and systematic in the specification of such defaults, and to ensure that the defaults do not interact with one another in unanticipated and undesirable ways. We therefore believe that this approach tends to yield a language where, although it may be possible to formulate common queries concisely, the language itself is complex and therefore difficult to understand and use.

Observation 10 *Defining a temporal language in terms of syntactic defaults is difficult since the non-temporal constructs do not offer a systematic and easy way to express the defaults.*

5.6 Approach V: Semantic Defaults

ATSQL introduces temporal statement modifiers to add temporal support to SQL [8, 4]. In contrast to syntactic defaults, statement modifiers are *semantic defaults* that indicate the intended semantics without specifying how to compute it.

The basic idea in statement modifiers is to offer a systematic means of constructing temporal queries from non-temporal queries, the motivation being that queries that are easily formulated in SQL on non-temporal relations are very difficult to formulate on temporal relations. With statement modifiers, one thus formulates a temporal query by first formulating the corresponding non-temporal query (i.e., assuming that there are no timestamp attributes on the argument relations) and then applies a statement modifier to this query.

For example, to formulate a temporal join the first step is to formulate the corresponding non-temporal join. Next, a modifier is prepended to express that temporal semantics are to be used. The modifier ensures that the argument timestamps overlap and that the resulting timestamp is the intersection of the argument intervals. If the enclosed query is simply a selection, the timestamps do not have to be transformed, and the only task of the modifier is to ensure that the original timestamps are returned as the timestamps of the result. If the enclosed statement is a difference, the modifier ensures that the intervals are appropriately subtracted.

Observation 11 *Statement modifiers are orthogonal to the SQL language and adding them to SQL is less understood than adding a new ADT.*

Unlike the languages that consider intervals as compact representations of sets of points, the use of statement modifiers makes it possible to give more meaning to the intervals. Thus, relations in ATSQL consist of interval timestamped tuples, and value-equivalent tuples with adjacent or overlapping intervals are permitted. Relation EMPL as given in Figure 3 is assumed in the following.

$Q_{\mathrm{ci}}^{\mathrm{ATSQL}}$**:** Query Q_{ci} is a temporal generalization of a non-temporal query. Thus, it can be formulated by prepending the non-temporal SQL query by the `seq vt` modifier:

```
seq vt
  select D, count(*) as Cnt
  from Empl
  group by D
```

$Q_{\mathrm{fi}}^{\mathrm{ATSQL}}$**:** By default, the `seq vt` clause operates at the lowest granularity and computes constant intervals. This behavior can be extended by allowing the user to specify different granularities or, in the general case, fixed intervals. Below we show an extended modifier that specifies the periods for which a result tuple is to be produced.

```
seq vt
for semesters(vtime(Empl))
  select D, count(*)
  from Empl
  group by D
```

In this query, the `semesters` function is a generative function that returns all semesters that a given interval timestamps spans, e.g., $\texttt{semesters}([2005/2, 2006/5]) = \{[2005/1, 2005/6], [2005/7, 2005/12], [2006/1, 2006/6]\}$.

$Q_{\mathrm{cum}}^{\mathrm{ATSQL}}$**:** Moving-window aggregation is an extension of regular instantaneous aggregation. Various syntactic constructs have been proposed for moving-window aggregation. We use the syntax of the Oracle OLAP extensions [] to illustrate how such aggregates can be incorporated into modifiers.

```
seq vt
with range
     between interval '3' months preceding
     and current
  select count(*)
  from Empl
  group by D
```

$Q_{\mathrm{nt}}^{\mathrm{ATSQL}}$**:** The last query must be evaluated independently of the timestamps of the argument tuples. This is achieved by using a `nseq vt` modifier (short for "non-sequenced valid time"), which indicates that what follows should be treated as a regular SQL query.

```
nseq vt
  select count(*) as Cnt
  from Empl
```

In summary, semantic defaults offer systematic support for writing temporal queries that can be evaluated on all sets of concurrent states of the argument relations in isolation. This language mechanism is independent of the syntactic complexity of the queries that the modifiers are applied to, which renders semantic defaults scalable across the constructs of the language being extended.

Observation 12 *Statement modifiers by and large separate the temporal and non-temporal parts of a query expression.*

While statement modifiers offer attractive means of formulating the example queries, it should be noted that extending a language with statement modifiers represents a much more fundamental change to the language than, e.g., extending the language with temporal abstract data types.

6 Summary

The temporal database research community has been quite prolific with respect to the design of new temporal query languages—a body of several dozen such languages exists. Based on the observation that many of these languages can be categorized according to the approach they take to providing temporal support, this paper investigates the support for temporal aggregation inherent to five such approaches.

More specifically, the paper initially characterizes temporal query languages according to whether they are point- or interval-based, noting that certain aggregation queries may be more difficult to formulate in point-based models. It then presents a general framework of temporal aggregation concepts. Building on this foundation and four example aggregation queries, the paper explores the aggregation capabilities of five distinct categories of temporal query languages. To make the coverage concrete, a prototypical query language serves as a representative for each approach.

The main findings are formulated in a number of observations. The paper affords an informal coverage of its subject in order to cover a wide range of concepts as well as to offer a foundation for further research.

The abstract data type approach is simple, but also very limited in the support offered. Its main strength is that adding ADTs to SQL is well understood, e.g., there exist ADTs for images, text, multimedia, etc. The main disadvantage is that advanced and systematic support for time-varying applications seems to require solutions that cannot be offered by extending SQL with new functions and predicates.

The fold/unfold approach enables easy conversion between point and interval timestamped representation of a relation. Using point timestamped relations makes the formulation of some queries easier, while interval timestamps are convenient for other queries, as well as for physical representation of relations and user display of query results. The fold/unfold approach is limited by being inherently point-based. The main strength is the conceptual simplicity of fold/unfold. On the downside the (syntactic) complexity of temporal queries remain fairly high and an efficient implementation of fold/unfold has yet to emerge.

The approach that solely uses point timestamps assumes that physical representation and display of relations are beyond the scope of the query language. This leads to a clean, point-based query language. Working solely with points greatly simplifies the formulation of instantaneous temporal queries. A possible drawback is that the user must frequently map between intervals and points since relations are represented in their compact form whereas statements are formulated against abstract databases. Also, some statements become system dependent. For example a count without duplicate elimination (e.g., `select count(X) from R`) returns the number of tuples in the abstract relation R. This number depends on the base granularity, which may differ among systems.

Next, with syntactic defaults, typical queries may be given very short formulations. However, it is challenging, if not impossible, to design a query language that systematically and comprehensively offers convenient syntactic defaults and that is also easy to understand. Syntactic defaults tend to not scale well since a complex syntactic default must be specified for a large number of constructs of the original language.

The notion of statement modifiers offers what may be

termed semantic defaults: modifiers are introduced that control the semantics of any query language statements. The strong point is the support for intervals and the systematic support for temporal queries that generalize snapshot queries. The approach by and large decouples the temporal and non-temporal parts in a statement. Thus, the presence of time-varying information does not change the formulation of the core query. A drawback is that this approach is new and that there are no experiences with such extensions to SQL.

7 Outlook

In step with the increasing digitization throughout society, the increasing networking of information systems, and the ability to store increasing amounts of data, increasing volumes of time-varying data are being accumulated and made available to users. Trends such as automatic data gathering using web-server logs in e-business applications and using sensors in a range of applications contribute to this development.

We are also witnessing an increase in analytical applications, often referred to as business intelligence applications, that extract useful information from large volumes of data, e.g., by means of aggregation. Thus, effective support for the formulation of aggregation queries is increasingly important.

In contrast, this paper's study indicates that the support for temporal aggregation in query languages is still lacking. In fact, the existing temporal query languages were largely designed with traditional record-keeping applications in mind, i.e., the kind of application found in banking where account-balances are kept for customers. In banking, a (constant) account balance is valid during a time interval, and it is valid for any subset of this time interval.

New applications that do not fit this rigid template are becoming increasingly important. This applies to applications that monitor or track continuous variables, e.g., positions of moving objects using GPS, the flow of water from a river into the sea, or temperature and humidity in different geographical locations. These applications typically rely on sampling, so they record values that are valid only for a single point in time. Values beyond these times must be interpolated or extrapolated. This type of scenario is also characterized by data uncertainty—values beyond the samples are not accurate, and even the samples may not be accurate.

Next, consider an attribute that records the accumulated rainfall in a certain location over a certain time interval. Unlike in the case of the account balance, a value of this attribute does not hold for any subset of the interval associated with it, as the accumulated rainfall in smaller interval is likely to be smaller. This type of attribute is a good example of attributes that carry semantics that differ from what is assumed by existing query languages.

These examples, which go beyond the setting assumed in this paper, illustrate that existing temporal query languages may be extended to offer much better support for temporal aggregation.

Finally, the increasing prominence of business intelligence has also brought new prominence to temporal aggregation. W. H. Inmon, known as the founder of data warehousing, mentions time variance as one of four salient characteristics of a data warehouse, and there is general consensus that a data warehouse is likely to exhibit a strong temporal orientation.

Being temporal, data warehouses are thus prime candidates to benefit from the advances in temporal aggregation. But cross-fertilization between temporal databases and data warehousing is lacking. In fact, some of the original impetus for a separate data model and query language for data warehouses arose from a perceived lack of temporal support in the relational model and SQL. Few attempts have been made to exploit the advances in temporal databases in the context of data warehousing, although notable exceptions do exist. The special dimensional data models used in data warehouses and the emphasis on supporting advanced query functionality bring novel challenges to temporal database research. For example, few attempts have been made at integrating temporal query languages with multidimensional query languages.

Acknowledgments

The work was partially funded by the Free University of Bolzano through the TTDBT project and the Municipality of Bozen-Bolzano through the eBZ-2015 initiative. C. S. Jensen is also an adjunct professor in Department of Technology, Agder University College, Norway.

References

[1] M. H. Böhlen, J. Gamper, and C. S. Jensen. An Algebraic Framework for Temporal Attribute Characteristics. *Journal of Annals of Mathematics and Artificial Intelligence*, 26 pages, to appear.

[2] M. H. Böhlen, J. Gamper, and C. S. Jensen. Multidimensional Aggregation for Temporal Data. In *Proc. EDBT*, pp. 257–275, 2006.

[3] M. H. Böhlen and C. S. Jensen. Temporal Data Model and Query Language Concepts. *Encyclopedia of Information Systems*, 4: 437–453, 2003, Academic Press.

[4] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Temporal Statement Modifiers. *ACM TODS*, 25(4): 407–456, 2000.

[5] I. T. Bowman and D. Toman. Optimizing Temporal Queries: Efficient Handling of Duplicates. *Data and Knowledge Engineering*, 44(2): 143–164, 2003.

[6] Y. Chen and P. Z. Revesz. Max-Count Aggregation Estimation for Moving Points. In *Proc. TIME*, pp. 103–108, 2004.

[7] J. Chomicki and P. Z. Revesz. Constraint-based Interoperability of Spatiotemporal Databases. *GeoInformatica*, 3(3): 211–243, 1999.

[8] J. Chomicki, D. Toman, and M. H. Böhlen. Querying ATSQL Databases with Temporal Logic. *ACM TODS*, 26(2): 145–178, 2001.

[9] J. Clifford and A. Tuzhilin (eds.). *Recent Advances in Temporal Databases: Proceedings of the International Workshop on Temporal Databases*. Workshops in Computing Series. Springer-Verlag 1995.

[10] O. Etzion, S. Jajodia, and S. Sripada (eds.). *Temporal Databases: Research and Practice*. LNCS 1399, Springer-Verlag 1998.

[11] C. S. Jensen and C. E. Dyreson (eds.). A Consensus Glossary of Temporal Database Concepts—February 1998 Version. [10, pp. 367–405].

[12] C. S. Jensen, and R. T. Snodgrass. Semantics of Time-Varying Information. *Information Systems*, 21(4): 311–352, 1996.

[13] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint Query Languages. *J. Comput. Syst. Sci.*, 51(1): 26–52, 1995.

[14] N. Kline and R. T. Snodgrass. Computing Temporal Aggregates. In *Proc. ICDE*, pp. 222–231, 1995.

[15] N. Kline, R. T. Snodgrass, and T. Y. C. Leung. Aggregates. In R. T. Snodgrass, editor, *The TSQL2 Temporal Query Language*, Chapter 21, pp. 395–425. Kluwer Academic Publishers, 1995.

[16] N. Kline and M. D. Soo. TIME-IT*: The* TIME *Integrated Testbed*, pre-beta version 0.1 available via anonymous ftp from `ftp.cs.arizona.edu`, 1995.

[17] A. C. Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *JACM* 29(3): 699–717, 1982.

[18] N. A. Lorentzos. The Interval-extended Relational Model and Its Application to Valid-time Databases. [37, pp. 67–91].

[19] N. A. Lorentzos and R. Johnson. Extending Relational Algebra to Manipulate Temporal Data. *Information Systems*, 13(3): 289–296, 1988.

[20] N. A. Lorentzos and Y. Mitsopoulos. Functional Requirements for Historical and Interval Extensions to the Relational Model. *Data and Knowledge Engineering*, 17(1): 59–86, 1995.

[21] N. A. Lorentzos and Y. G. Mitsopoulos. SQL Extension for Interval Data. *IEEE TKDE*, 9(3): 480–499, 1997.

[22] L. E. McKenzie Jr. and R. T. Snodgrass. Evaluation of Relational Algebras Incorporating the Time Dimension in Databases. *ACM Computing Surveys*, 23(4): 501–543, 1991.

[23] B. Moon, I. F. Vega Lopez, and V. Immanuel. Efficient Algorithms for Large-Scale Temporal Aggregation. *IEEE TKDE*, 15(3): 744–759, 2003.

[24] G. Özsoyoğlu and R. T. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE TKDE*, 7(4): 513–532, 1995.

[25] P. Z. Revesz. *Introduction to Constraint Databases*. Springer, 2002.

[26] P. Z. Revesz. Efficient Rectangle Indexing Algorithms Based on Point Dominance. In *Proc. TIME*, pp. 210–212, 2005.

[27] P. Z. Revesz, R. Chen, P. Kanjamala, Y. Li, Y. Liu, and Y. Wang. The MLPQ/GIS Constraint Database System. In *Proc. SIGMOD*, p. 601, 2000.

[28] P. Z. Revesz and Y. Chen. Efficient Aggregation over Moving Objects. In *Proc. TIME*, pp. 118–127, 2003.

[29] J. F. Roddick and J. D. Patrick. Temporal Semantics in Information Systems—a Survey. *Information Systems*, 17(3): 249–267, 1992.

[30] R. T. Snodgrass (ed.). *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*, 1993.

[31] R. T. Snodgrass. Temporal Object Oriented Databases: A Critical Comparison. Ch. 19, pp. 386–408, of W. Kim, *Modern Database Systems: The Object Model, Interoperability and Beyond*. Addison-Wesley/ACM Press 1995.

[32] R. T Snodgrass (ed.), I. Ahn, G. Ariav, D. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Käfer, N. Kline, K. Kulkarni, T. Y. Leung, N. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo, and S. M. Sripada. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers 1995.

[33] R. T. Snodgrass. Temporal Databases. Part II of C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems*. Morgan Kaufmann Publishers 1997.

[34] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann Publishers 2000.

[35] R. T. Snodgrass, I. Ahn, G. Ariav, D. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Käfer, N. Kline, K. Kulkarni, T. Y. C. Leung, N. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo, and S. M. Sripada. TSQL2 Language Specification. *ACM SIGMOD Record*, 23(1): 65–86, 1994.

[36] R. T. Snodgrass, S. Gomez, and E. McKenzie. Aggregates in the Temporal Query Language TQuel. *IEEE TKDE*, 5(5): 826–842, 1993.

[37] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. T. Snodgrass (eds.). *Temporal Databases: Theory, Design, and Implementation*. Benjamin/Cummings Publishers 1994.

[38] C. I. Theodoulidis and P. Loucopoulos. The Time Dimension in Conceptual Modelling. *Information Systems*, 16(3): 273–300, 1991.

[39] D. Toman. Point-Based Temporal Extensions of SQL and Their Efficient Implementation. [10, pp. 211–237].

[40] Y. Wu, S. Jajodia, and X. S. Wang. Temporal Database Bibliography Update. [10, pp. 338–366].

[41] J. Yang and J. Widom. Incremental Computation and Maintenance of Temporal Aggregates. *The VLDB Journal*, 12(3): 262–283, 2003.

[42] D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopoulos, and B. Seeger. Efficient Computation of Temporal Aggregates with Range Predicates. In *Proc. PODS*, pp. 237–245, 2001.