Aggregates in the Temporary Query Language TQuel

Richard T. Snodgrass, Santiago Gomez, and L. Edwin McKenzie, Jr.

***Abstract*— This paper defines new constructs to support aggregation in the temporal query language TQuel and presents their formal semantics in the tuple relational calculus. A formal semantics for Quel aggregates is defined in the process. Multiple aggregates; aggregates appearing in the where, when, and valid clauses; nested aggregation; and instantaneous, cumulative, moving window, and unique variants are supported. These aggregates provide a rich set of statistical functions that range over time, while requiring minimal additions to TQuel and its semantics. We show how the aggregates may be supported in an historical algebra, both in a batch and in an incremental fashion, demonstrating that implementation is straightforward and efficient.**

***Index Terms*— Aggregate, correlation query, moving window aggregate, Quel, query language, temporal database, temporal partitioning, TQuel, tuple calculus, valid time.**

I. Introduction

Aggregate operators in relational database query language compute a scalar froma collection of tuples. Most commercially available relational database management systems (DBMS's) provide several aggregate operations [10], [13], [24], [38], [54]. Recently attention has been focussed on *temporal databases* (TDB's) that represent the progression of states of an enterprise over time. We have developed a new language, TQuel (*Temporal QUEry Language*), to query a TDB [47]. TQuel is a derivative of Quel [21], query language for the Ingres DBMS [50]. TQuel was designed to be a minimal extension, both syntactically and semantically, for that language. Since Quel is fairly comprehensive in its support of aggregates, a goal in the TQuel design was to extend those aggregates to operate over temporal relations.

This paper defines and formalizes aggregaes in TQuel. We begin in Section II by describing the Quel aggregates. An intuitive introduction to the TQuel aggreggates is given in Section III. The resulting language subsumes all aspects of aggregates appearing in other proposals. Section IV is devoted to a formal semantics of Quel aggregates. As the core of the retrieve statement and the modification statements were previously formalized in [54] and [47], respectively, this completes the formal definition of Quel. Section V extends these semantics to TQuel. The result is a complete formal semantics for TQuel and its snapshot subset Quel. A complete formal semantics for no other relational query language, conventional or temporal, has been defined. We then examine how aggregates may be supported in an historical algebra, demonstrating that techniques for processing conventionl aggregates may be extended in a straightforward manner to process temporal aggregates, and that incremental evaluation of temporal aggregates is available for improved efficiency. The final section compares TQuel aggregates with those of several other query languages supporting time.

Manuscript received May 7, 1992. This work was supported in part by NSF grants DCR-8402339 and IRI-8902707, by ONR contract N00014-86-K-0680, and by a Junior Faculty Development Award from the UNC-CH Foundation. The first author was supported in part by an IBM Faculty Development Award. The third author was sponsored in part by the U.S. Air Force.

R. T. Snodgrass is with the Department of Computer Science, University of Arizona, Tuscon, AZ 85721.

S.L. Gomez is with Centro Nacional de Computacion, Asuncion, Paraguay.

L.E. McKenzie, Jr. is with AFCAC/SY, Hanscom Air Force Base, MA 01731.

IEEE Log Number 9211306.

TABLE I

Name	*Rank*	*Salary*
Jane	Full	44000
Merrie	Associate	40000

II. Aggregates in Quel

In this section we present an informal specification for the Quel aggregates. The Quel operations for aggregation are `count`, `any`, `sum`, `avg`, `min`, and `max`. Thsese operators can be used in two types of aggregation.

1) *Scalar aggregates*, yielding a *single value* as the result.
2) *Aggregate functions*, producing several values determined by calculating the aggregate over a subset of the relation. Each subset consists of the tuples such that the contents of one or more attributes grouped in a by-list are the same. Hence the result of an aggregate function is a *relation* whose number of tuples equals the number of different values in the by-list. (Since a scalar aggregate is in fact a function, this terminology is confusing: we adhere to it only because it has become established [9]).

While scalar aggregates are independent of the query in which they are nested, aggregate functions are not. Since each value computed by such a function carries information on part of a relation, tuple variables in the by-list must be linked to the corresponding tuple variables, if any, in the *outer query*—that is, they should refer to the same part of the relation. (The *inner query*, as opposed to the outer query, is the one consisting of the attribute to be aggregated, the by-list, and the inner where clause.)

Aggregation performed over the set of strictly different values in an attribute is called unique aggregation. Quel supports three unique aggregates: `countU`, `sumU`, and `avgU`. Unique versions of `any`, `max` and `min` are not necessary.

As an example, suppose the relation *Faculty* holds relevant data, say name, rank and salary, about the professors in a university department (see Table I).

Rank	*NumFaculty*	*NumRanks*	*NumInRank*
Full	2	2	1
Associate	2	2	1

Fig. 1. Result of a query containing aggregates.

The following query computes the number of faculty members, the total numbers of ranks, and the number of faculty members at each rank.[1]

```
range of f is Faculty
retrieve (f.Rank,
          NumFaculty=count(f.name),
          NumRanks=countU(f.Rank),
          NumInRank=count
                   (f.Name by f.Rank))
```

The range statement declares a typle variable f that will be associated with the *Faculty* relation. The retrieve statement contains the target list of attributes to be derived for the output relation (attributes are denoted by a tuple variable followed by a period followed by the attribute name). The output relation (Fig. 1) contains as many tuples as actual values exist in the by-list. If there had been no by-list, NumInRank would be 2 in all the derived tuples. Also note that the NumFaculty and NumRanks values are independent of the Rank, since a by-clause was not used in their definition.

By their very nature, both scalar aggregates and aggregate functions operate on the entire relation. However, they can be *locally* restricted via a where clause to operate only on certain tuples of the relation. The local or inner where clause is processed separately from the outer one of the query.

III. Temporal Aggregates in TQuel

In the previous section we have seen the various Quel aggregates. We now introduce TQuel aggregates in an intuitive way through examples. We first give an overview of the TQuel language and then turn to aggregages.

TQuel is an extension of Quel, augmented to handle the time dimension [47]. TQuel supports valid, transaction, and user-defined time, and thus supports temporal queries [46]. Of the three, valid time, modeling the real world occurrence of an event, is by far the hardest to support in aggregates. Transaction time, modeling the storage of information in a database, may be supported through one additional term in the tuple calculus semantics. User-defined time, an encoding whose semantics is maintained by application programs, is handled in an identical manner to more conventional data types such as integers and character strings; all that is necessary are input, output, and comparison functions. To simplify the exposition, we will not use transaction or user-defined time in the example queries or in their formal semantics. In the general formal semantics, we will include transaction time, to illustrate how easy it is to support.

Temporal relations are four dimensional. Multiple tuples containing multiple attribute values contribute two dimensions; valid and transaction time contribute the other two dimensions. For both the examples and the semantics, we embed these four-dimensional structures into two dimensional tables, appending additional, implicit time attributes (generally two valid attributes, *from* and *to*) that are not directly accessible to the user. Other embedding are possible (five are given in [47]), but will not be used here. The degree (*deg*) of a temporal relation is the number of explicit attributes.

The TQuel retrieve statement augments the standard Quel retrieve statement by including

- a *when* clause, paralleling the already existing where clause, to select tuples whose temporal attributes satisfy desired temporal constraints;
- a *valid-at* clause that permits the assignment of a non-default and possibly computed value to the valid-time attribute of a target event relation;
- *valid-from* and *valid-to* clauses that permit the same kind of assignment to the valid-time attributes of a target interval relation; and
- an *as-of* clause to specify rollback to a previous transaction or series of transactions.

A. Adding Aggregates to TQuel

In defining aggregates in TQuel, we kept several goals in mind. First, TQuel aggregates should include all of Quel's aggregates, so that TQuel remains a strict superset of Quel. Second, the snapshot reducibility of TQuel to Quel (proven elsewhere [47]) should be maintained, so that the TQuel version of a Quel aggregate will perform the same fundamental operation. This will ensure that the intuitive semantics of Quel applies to TQuel. This goal impacts the design of both the syntax and the semantics of the new constructs. Third, the most needed strictly temporal aggregates, including those that evaluate to scalar values and those that evaluate to time-stamps, should be provided. Fourth, the semantics should be independent of the time-stamp granularity. Finally, features introduced in other temporal query languages should also be available in TQuel, and be accommodated in its formal semantics.

There are some differences between Quel and TQuel aggregates. Historical and temporal databases are characterized by the changing condition of their relations: at time t_1 a relation contains a set of tuples, and at time t_2 the same relation may contain a different set. Since aggregates are computed from the entire relation, this in turn causes the value of an aggregate to change from, say, v_1 to v_2. Hence, while in Quel an aggregate with no by-list (scalar aggregate) returns a single value, in TQuel the same aggregate returns, generally speaking, a *sequence* of values, each associated with its valid time. For an aggregate with a by-list, a sequence of values for each value in the by-list is generated.

Let us apply the example query on the historical relation in Table II (since TQuel is a superset of Quel, that query is a valid TQuel query).

With the default when clause (**`when`** `f` **`overlap now`** and valid clause (**`valid from begin of`** `f` **`to end of`** `f`), the example query

[1]Thoughout the paper, a fixed-width font is used for operators in the query language (e.g., `count`); a bold, fixed-width font is used for keywords (e.g. **`year`**); and italics is used for functions in the formal semantics (e.g., *count*).

TABLE II

Name	*Rank*	*Salary*	*from*	*to*
Jane	Assistant	25000	9-71	12-76
Jane	Associate	33000	12-76	11-80
Jane	Full	34000	11-80	12-83
Jane	Full	44000	12-83	∞
Merrie	Assistant	25000	9-77	12-82
Merrie	Associate	40000	12-82	∞
Tom	Assistant	23000	9-75	12-80

```
range of f is Faculty
retrieve (f.Rank,
          NumFaculty=count(f.Name),
          NumRanks=countU(f.Rank),
          NumInRank=count
                 (f.Name by f.Rank))
```

would result in the relation in Table III, which is identical to that evaluated by Quel, shown in Fig. 1.

The default requests the *current* value for the aggregate. Defaults are discuss in detail elsewhere [48]; these defaults must be defined carefully to ensure snapshot reducibility. To extract the *history* of the requested count, simply use an explicit when clause: **when true**. The altered query yields the tuples in Table IV.

The `count` may change only when a Faculty tuple is created, or becomes invalid. As can be seen, for each rank there can be more than one related count over time.

Quel allows an inner where clause to preselect tuples for the computation of the aggregate; otherwise, aggregates always operate on the entire relation. Similarly, in TQuel the inner where, when, and as-of clauses serve the same purpose. An inner valid clause is not allowed, because the interval of validity for the value calculated by the aggregate is indirectly specified using the for clause, to be discussed shortly.

The above example illustrates our approach to computing TQuel aggregates. To aid in understanding temporal aggregation, we now present one possible way to compute an aggregate over a given attribute of relation R. Note that this description is at a logical level; the implementation is free to perform aggregation in any manner that is consistent with the semantics to be presented later.

1) Determine the periods of time during which R remained "constant" that is, no new tuples entered the relation (and, if R is an interval relation, no tuples became invalid).
2) For each constant set of tuples in R, select the tuples that satisfy all the qualifications required by the inner where, when, and as-of clauses, if any. Defaults are used if those clauses are not present.
3) If there is a by-list with this aggregate, partition each constant set of tuples into subsets, each subset corresponding to one value of the by-list attributes. Each group of selected tuples is called an *aggregation set.*
4) Compute the aggregate for each aggregation set, producing a single value.
5) Associate the result with each combination of tuples participating in the original query, with the aggregation set selected a) using the values indicated in the by-clause, b) using the valid time of the underlying aggregation set, and c) using the interval or event specified in the valid clause.

TABLE III

Rank	*NumFaculty*	*NumRanks*	*NumInRank*	*from*	*to*
Full	2	2	1	12-83	∞
Associate	2	2	1	12-82	∞

TABLE IV

Rank	*NumFaculty*	*NumRanks*	*NumInRank*	*from*	*to*
Assistant	1	1	1	9-71	9-75
Assistant	2	1	2	9-75	12-76
Assistant	2	2	1	12-76	9-77
Associate	2	2	1	12-76	9-77
Associate	3	2	1	9-77	11-80
Assistant	3	2	2	9-77	12-80
Full	3	2	1	11-80	12-80
Full	2	2	1	12-80	∞
Assistant	2	2	1	12-80	12-82
Associate	2	2	1	12-82	∞

The basic strategy consists of reducing a TQuel aggregate to a series of Quel-style aggregates, each applied on a period of time when the relation does not change its contents. Each value of the aggregate is associated with an assignment of values to the by-list attributes, and is attached to the particular period of time it was valid. At each point in time, there is exactly one value of the aggregate for each combination of values of the by-list attributes.

This approach is necessarily more complex than that for Quel aggregates. In TQuel, for each interval during which all base relations participating in the aggregate(s) remain "fixed," an aggregate tuple is computed for each aggregation set. In Quel, all base relations are already fixed, since the relations do not vary over time. This aggregate tuple, along with tuples from the base relations that are valid over the interval, determine the output tuples for the interval. Whereas Quel uses only the explicit attribute values via the by-clause to connect the aggregate tuple with the participating tuples in the retrieve statement, TQuel also uses the implicit time values. Any combination of aggregate and base-relation tuples that satisfy all qualifications required by the outer where and when clauses, and also overlap, produce an output tuple. In addition, the valid time of each output tuple is required to be the overlap of the interval or event specified by the valid clause with the overlap of the aggregate tuple and base-relation tuples named in the aggregate.

The restriction that the valid time of the output tuple be the intersection of the valid times of some of the participating tuples and the aggregate tuple as well as the time specified by the valid clause does not limit the range of queries that TQuel can support. To support queries whose output is derived from aggregate and base-relation tuples valid over different intervals, we can simply pre-compute the aggregates and treat them as ordinary historical relations in the main TQuel query.

B. Cumulative versus Instantaneous Aggregates

An aggregate may or may not take into account tuples that are no longer valid. The following definitions are useful [27].

Cumulative Aggregate: an aggregate whose value for each point *t* in time is computed from all tuples that have been valid in the past, as well as those valid at *t*.

Instantaneous Aggregate: an aggregate whose value for each point *t* in time is computed only from the tuples valid at time *t*.

These aggregates act differently when applied to an event or an interval relation. For an event relation, as the length of the time unit (the time-stamp granularity) is reduced, the probability of finding any valid tuples decreases. Aggregates such as `count`, applied at a given instant, would thus return different results depending upon the granularity of valid time. On the other hand, it is always possible to count the events that have occurred in the past, or in a given period of time, in a cumulative fashion. For an interval relation, tuples are valid over an interval of time which is at least as long as the time-stamp granularity, and therefore the above problem does not exist. We therefore restrict aggregate operators over event relations to be cumulative, while aggregate operators over interval relations can have both an instantaneous and a cumulative version. However, each value of an aggregate, be it instantaneous or cumulative, is valid during a period of time.

For cumulative aggregates, the user must specify how far in the past to include tuples used to compute a value at time *t*. The for clause is used for this purpose. Instantaneous aggregates (the default) are specified using **`for each instant`**. If all previous tuples are to participate, **`for ever`** is used. Intermediate cases, such as using only those tuples valid at some point in the previous year, are specified using **`for each`** < span >, e.g., **`for each year for each day`**. If, say, `count (`**`for each year`**`)` is used, then the aggregate, when computing a value valid at a particular month m, will operate over all tuples that were valid sometime during the year up to and including the month m. The value at 3-76 will include all tuples valid sometime during 4-75 through 3-76; the value at 4-76 will include the (potentially different) tuples valid sometime during 5-75 through 4-76. The interval used (in this case, year) is termed the *window,* and such aggregates are termed *moving-window aggregates.* Such aggregates were first proposed in TSQL [36].

Fig. 2, which shows the results of the following query, illustrates the difference between the various kinds of aggregates on an interval relation. We have specified **when true** to obtain the entire history of the counts.

```
retrieve
   (C1=count(f.Rank for each instant),
    C2=count(f.Rank for each year),
    C3=count(f.Rank for ever),
    C4=countU(f.Rank for each instant),
    C5=countU(f.Rank for each year),
    C6=countU(f.Rank for ever))
   when true
```

Note that the values associated with **for each year** (e.g., C2) are in a sense between the values associated with **`for each instant`** (e.g., C1) and **`for ever`** (e.g., C3). The value at 6-81 associated with **`for each instant`** counts an Assistant Professor (Merrie) and a Full Professor (Jane), for a total of 2; **`for each year`** counts 2 Assistants (Merrie and Tom), one Associate (Jane), and one Full (also Jane, since her promotion occurred within the year before 6-81), for a total of 4; and **`for ever`** counts 3 Assistants (Jane, Tom, and Merrie), one Associate (Jane), and one Full (Jane), for a total of 5. The unique aggregate **`for each year`** (C5) counts one Assistant (Tom *or* Merrie), one Associate (Jane), and one Full (also Jane), for a total of 3, since all three ranks were represented over the previous year. The values associated with **`for ever`** are monotonically increasing.

C. New Aggregates

All Quel aggregates have a TQuel counterpart. There are also some aggregates unique to TQuel. The first, `stdev`, which computes the standard deviation, is quite similar to `avg`, applying both to snapshot relations and temporal relations. The remaining new aggregates are strictly temporal.

Quel's aggregates may be classified as a) select a particular value from the underlying relation (e.g., `min` and `max`); b) compute a new value of the domain of the attribute from the values in the underlying relation (e.g., `avg` and `sum`); and c) compute a non-dimensional quantity (e.g., `count` and `any`). For temporal aggregates, these three generalize directly into five categories; TQuel aggregates exist in each. In the first category, aggregates *select* a value from the underlying relation based on time.

`first` This aggregate returns, at each point in time, the oldest value of the given attribute, that is, the one associated with the first valid tuple. If two tuples have the same *from* value, one is arbitrarily selected.

`last` This aggregate is analogous to first.

One could also envision an aggregate to select the ith occurring interval, for a given i such an aggregate was proposed in HQuel [17].

Aggregates in the second category *compute* a new value of the domain of the attribute from the values of the underlying relation, based on time.

`rate` This aggregate computes the average growth or decrease experienced by values of an attribute over time. This aggregate is only applicable to numeric attributes in event relations. It returns a value indicating growth per time unit, e.g., feet/hour, or dollars/month. The time unit can be optionally specified by the user by means of the **per** clause (see the syntax in the appendix): **`per hour, per month, per 3 months`**. This aggregate compares the attribute value of each tuple with the attribute value of its chronologically previous tuple, relative to the time elapsed, and smooths the comparisons by taking their arithmetic mean. This aggregate was first proposed by Tansel and Arkun for HQuel [51]. This aggregate is useful in statistical time series analysis.

Aggregates in the third category compute a nondimensional quantity, based on time.

`var` *VAR*iability of time spacing: the degree of inequality of

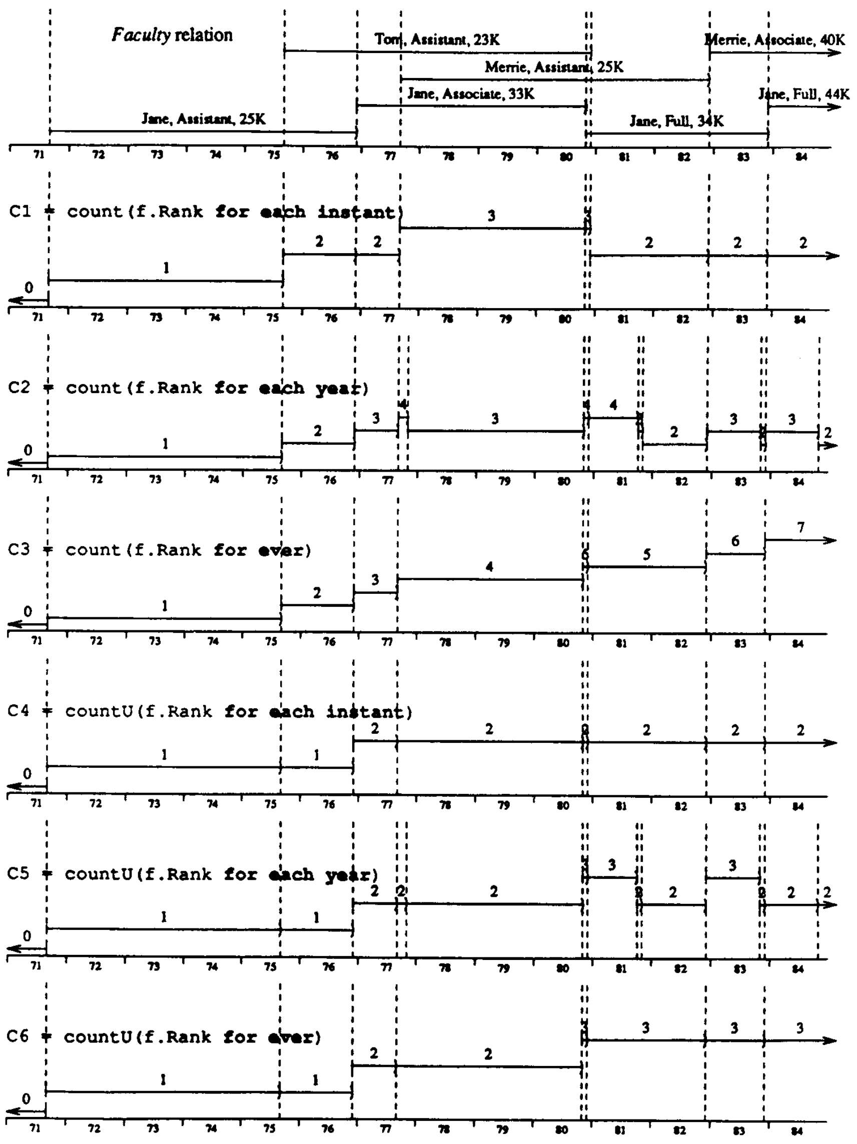


Fig. 2. Comparison of six aggregate variants.

TABLE V

Stock	*Price*	*at*
DCE	127	9:00
NRC	40	9:00
NRC	35	9:05
DCE	125	9:15
NRC	37	9:15
NRC	39	9:20
DCE	122	9:30
NRC	40	9:30
NRC	41	9:35
NRC	40	9:40
DCE	121	9:45
DCE	120	10:00
NRC	39	10:00
NRC	38	10:05
DCE	118	10:15
NRC	39	10:15

TABLE VI

Stock	*Price*	*VarSpacing*	*GrowthRate*	*from*	*to*
NRC	37	0.33	−0.20	9:05	9:15
NRC	39	0.35	0.27	9:05	9.20
NRC	40	0.33	0.20	9:05	9:30
NRC	41	0.35	0.13	9:05	9:35
NRC	39	0.57	0.00	10:05	10:15

the time spacing within a given set of events (the argument to this aggregate is an event expression evaluating to an event). This aggregate returns a nondimensional quantity which has the same value for each attribute. A value of 0 indicates the tuples are perfectly spaced. This aggregate also considers the tuples in chronological order. It finds the ratio of the standard deviation of the time lengths from one tuple to the next, to the average of those time lengths. A ratio is used to ensure that the measure is independent of time-stamp granularity. This aggregate is useful in statistical time series analysis.

The last two categories consist of aggregates that evaluate to valid time. Aggregates in the fourth category select events or intervals from the events or intervals in the underlying relation.

`earliest` The oldest time period of an interval relation, that is, the first *from-to* interval, or the oldest event, that is, the first *at* event. If two tuples of an interval relation have the same *from* value, the one with the earlier *to* time is considered to be older.

`latest` This aggregate is analogous to earliest.

An aggregate in the fifth category computes a new time from events in the underlying relation.

`rising` The maximal interval culminating in the final event of the underlying relation in which values of all events occurring at a particular time are greater than or equal to values of all events occurring immediately previously. If it is applied to an interval relation, it uses only the starting times.

The requirement that *all* events be rising may seem overly restrictive. However, by combining the operator with other constructs, the restrictions may be effectively relaxed. First, if a by clause on a key is used, then there will be only one value valid at any time. Applying the aggregate to the value of min or max, as in `rising (max(A.price))`, also ensures only one value valid at a time. Finally, determining the interval when the value is falling is easily done by applying the aggregate to the inverse, as in

```
rising(min(−A. price)).
```

These last three aggregates are called *aggregated temporal constructors* because they return a time interval as their result. They can be employed by the user to specify conditions in the temporal qualification (when clause) or the valid time (valid clause). To adhere to the syntax of temporal expressions and predicates, these aggregates take an interval expression, rather than a scalar valued expression, as an argument.

We give one example here that uses the `var`, `rate`, and `rising` aggregates. Examples that employ the other new aggregates, and that demonstrate nested aggregation, aggregates appearing outside the target list, and a when clause within an aggregate, are given elsewhere [48].

This example references the event historical relation

stocks, (Stock, Price):

containing the tuples in Table V.

The following query determines, for those stocks that have been rising in price, how equally spaced the quotations are in time, and how fast the price grew over the previous fifteen minute interval.

```
range of s is stocks
retrieve(
    Stock = s.stock, Price = s.Price,
    VarSpacing = var
                 (x for ever by s.Stock),
    GrowthRate = rate
                 (s.Price per minute
                 for each 15 minutes by s.
                 Stock))
valid from begin of rising (s. Price by s.
Stock) to s when true
```

Since we want the history, we override the default when clause. The result is the relation in Table VI.

The interval indicates how long the stock had been rising. The *Price* and the *VarSpacing* apply to the terminating event. Note that all of the aggregates are computed on a per-stock basis. The DCE stock does not appear because its price never rises.

Computation of the variability of time spacing, for any attribute, consists of a) sorting the relevant tuples by their *at* attribute and b) considering every pair of chronologically consecutive tuples, S_i and S_{i+1}, finding the coefficient of variation of the length of time from event S_i to event S_{i+1}, that is,

$$\frac{standard\ deviation\ of\ < S_2[at] - S_1[at], \cdots, S_{i+1}[at] - S_i[at] >}{average\ of\ < S_2[at] - S_1[at], \cdots, S_{i+1}[at] - S_i[at] >}$$

The values of *VarSpacing* in the first four tuples is fairly small because the intervening interval for the NRC stock oscillates between 5 and 10 minutes. For the last tuple, the *VarSpacing* almost doubles, due to the anomalous 20 minute interval ending at 10:00. Incidentally, the *VarSpacing* for the DCE stocks are all 0, because the records are precisely spaced at 15-min intervals.

To compute the rate, we a) again sort the tuples by their *at* attribute, and b) for each pair of chronologically consecutive tuples S_i and S_{i+1}, compute the increment of the value $S_{i+1}[Yield] - S_i[Yield]$, averaged over previous pairs (**for each** 15 **minutes**), and then normalize over a minute (**per minute**). The *GrowthRate* at 9:15 is negative even thought the stock's price is rising then because the net effect over the previous 15 min was a drop in price (from 40 to 37). The GrowthRate at 10:15 is 0 because the price at the end of the interval (39) is identical to the price at the beginning of the interval; nevertheless, at 10:15 the stock's price was rising. The rate is greatest at 9:20, when the stock experienced an increase of 4 points over the previous 15 minutes.

D. Defaults

Defaults must be chosen carefully to maintain the snapshot reducibility to Quel, thereby allowing TQuel aggregates to be used in exactly the same way as Quel aggregates. Each default may be overridden with the explicit use of the clause. There are two places where default clauses may apply: the outer retrieve statement and within the aggregate. The default clauses in the outer retrieve statement without aggregates was given in [47].

```
valid from begin of
    (t_k overlap ··· overlap t_k)
    to end of(t_1 overlap ··· overlap t_k)
```

```
where true
when $t_1$ overlap ··· overlap $t_k$
as of now
```

where $t_1, \cdots, t_k$ are the tuple variable appearing in the query.

When aggregates are included in the query, we must distinguish between the tuple variables appearing inside and outside the aggregate. Tuple variables are included in the default when and valid clauses only if they appear outside an aggregate. If no tuple variable appears outside an aggregate, the default clauses are as follows.

```
valid from
valid from beginning to forever
where true
when true
as of now
```

The following defaults are assumed within each non-temporal aggregate, and are quite similar to the defaults used in the outer query.

```
for each
for each instant
where true
when $t_1$ overlap ··· overlap $t_k$
as of $\alpha$ through $\beta$
```

where $t_1, \cdots, t_k$ are the tuple variables appearing in the aggregate, and α and β are the expressions (or their defaults) appearing in the retrieval statement itself. The temporal aggregates differ in that their default for clause is **for ever**. The default per clause is the span specified in the for clause. If the for clause is not specified, or is **for ever**, then the rate aggregate must have an explicit per clause.

IV. Tuple Calculus Semantics of Quel Aggregates

Our approach to the semantics is based on Klug's method, which was used in a separate, more formal tuple relational calculus [30]. In this approach, each aggregate is associated with a function. This function is applied to a set of r-tuples, resulting in a single tuple containing r attribute values, with each attribute value equivalent to applying the aggregate over that attribute. By applying the function to the set of complete tuples, the distinction between unique and non-unique aggregation can be preserved.

Let R be a relation of degree r containing n and $n \geq 0$, let t be a tuple variable associated with R. For example, associated with the count aggregate is the function *count*$(R) \triangleq (n, \ldots, n)$, which yields a tuple whose r components equal n. The functions for the remaining Quel aggregates have similar definitions, and are given elsewhere, as are restrictions on the domains required by the aggregates [48].

These functions are used in the tuple calculus semantics. Let F be any of the aggregates defined in Section II and III-A. Quel queries with one aggregate function in the target list are of the form

```
range of $t_1$ is $R_1$
...
range of $t_k$ is $R_k$
retrieve ($d_1.a_1, \ldots, d_j.a_j$,
            $y = F(e_1.b_1$ by $e_2.b_2, \cdots, e_l.b_l$ where $\psi_1$))
   where $\psi$
```

in which

$$1 \leq i \leq j, d_i \in \{t_1, \cdots, t_k\}$$
$$1 \leq i \leq l, e_i \in \{t_1, \cdots, t_k\}$$
$$1 \leq i \leq j, a_i \in Attr(d_i)$$
$$1 \leq i \leq l, b_i \in Attr(e_i)$$

where $Attr\ (d_i)$ is the set of attributes associated with the relation associated with the tuple variable d_i Although values in a target list can be expressions, rather than simply attributes, we ignore that detail here for simplicity of notation. There is also the restriction that the tuple variable(s) mentioned in ψ_1 must be either e_1 or one of the tuple variables appearing in the by-clause: $e_2, \cdots, e_l$. Otherwise, there may be many more tuples participating in the aggregate, i.e., those from additional tuple variables, thereby generating unexpected results from the aggregate. The attributes outside the aggregate, $a_1, \cdots, a_j$, and the attributes used within the aggregate, $b_2, \cdots, b_l$ usually overlap, but need not.

Informally, this aggregate

1) gathers all combinations of tuples from the relations associated with the tuple variables appearing in the aggregate,
2) removes all resulting tuples that do not satisfy the condition in the where clause of the aggregate,
3) partitions the resulting tuples by the values of the attributes listed in the by-clause,
4) applies the aggregate to each partition,
5) and finally associates the result with each combination of tuples participating in the original query that satisfy the outer where clause, with the partition selected using the values indicated in the by-clause.

We specify the partition of the relations associated with the tuple variables appearing in the aggregate. Initially assume that the tuple variables $e_1, \cdots, e_l$ are all distinct. Let us first consider the case where no by-clause is present. The aggregate is applied to the following set.

$$P \triangleq \left\{ e_1^{(p)} \mid (R_{e_1}(e_1) \wedge \psi_1') \right\}$$

where R_{e_1} is the relation associated with tuple variable e_1 and $p \triangleq \deg(R_{e_1})$ is the *degree* of R_{e_1}, that is, the number of attributes in each tuple of R_{e_1}. In the tuple calculus, $R(e)$ states that e is a tuple in R. We use ψ_1' instead of ψ_1 to indicate modifications for attribute names and Quel syntax conventions. For the first aggregate in the example, `count(f.Name)`, there is no where clause (the default is **where true**).

$$P^1 \triangleq \left\{ f^{(3)} \mid Faculty(f) \wedge true \right\}$$
$$= \{(\text{Jane, Full, 40000}), (\text{Merrie, Associate, 40000})\}$$

Let F be the aggregate operator defined above corresponding to the Quel aggregate `F` (e.g., if `F` is `count`, F is *count*). A term of the form $F(R)$ will denote the tuple obtained from the application of aggregate operator F to relation R. The operator F applies the same aggregate to every attribute in R. Let $F(P)[m]$ denote the m-th attribute of the tuple evaluated by $F(P)$. For the example query, $count(P^1) = \{(2,2,2)\}$ and $count(P^1)[Name] = 2$.

The aggregates as defined cannot do unique aggregation directly, because they operate on relations, not on attributes. It turns out, however, that a slight change of the partition solves the problem. Let the modified partition be defined in terms of P as

$$U \triangleq \left\{ u^{(1)} \mid (\exists w)(w \in P \wedge u[1] = w[b_1]) \right\}$$

with b_1 being the attribute over which the aggregate is performed. Attribute values that are components of tuples may be selected in two ways in the tuple calculus: with brackets enclosing the index of the attribute, e.g., $u[1]$, or with brackets enclosing the name of the attribute, e.g., $w[b_1], w[\text{Rank}]$. The net effect of this is the elimination of all duplicate values from the attribute upon which aggregation will be performed. The tuple calculus semantics of unique aggregates is simply obtained by substituting U for P in the main formula of the previous section, and using the previously defined operators *count, sum,* and *avg.* For the countU aggregate of the example,

$$P^2 = \left\{ f^{(3)} \mid Faculty(f) \right\} = Faculty$$

$$U^2 = \left\{ u^{(1)} \mid (\exists w)(w \in P^2 \wedge u[1] = w[\text{Rank}]) \right\}$$
$$= \{(\text{Associate}), (\text{Full})\}$$

When *count* is applied to this set, the result is 2.

When a by-clause is present, as in the third aggregate in the example, `count(f.Name by f.Rank)`, we must partition the set and apply the aggregate to each partition. To ensure that the correct partition is used in the primary tuple calculus expression, we label each of the partitions with the attribute values used to define it. Define a partition P on the underlying relations named by the aggregate in the query as a collection of sets of tuples, with each set identified by $n-1$ values $a_2, \cdots, a_l$.

$$P(x_2, \cdots, x_l) \triangleq \left\{ u^{(p)} \mid (\exists e_1) \cdots (\exists e_l)(R_{e_1}(e_1) \right.$$
$$\wedge \cdots \wedge R_{e_l}(e_l)$$
$$\wedge\, u = e_1 \,\|\, \cdots \,\|\, e_l$$
$$\wedge\, e_2[b_2] = x_2 \wedge \cdots \wedge e_l[b_l] = x_l$$
$$\left. \wedge\, \psi_1') \right\}$$

where $\|$ denotes concatenation and $p \triangleq \sum_{i=1}^{n} \deg(R_{t_i})$. If there is no by-clause, then P is a set of p-tuples over which the aggregate is to be applied, as discussed above. Otherwise, each of the combinations of values $x_2, \cdots, x_l$ of attributes appearing in the by-clause produces one partition on which the aggregate has to be applied. The tuple calculus statement will supply each combination of values existing in the attributes specified by the by-clause, as will be seen shortly. One can verify from the definition that every tuple in the cross product of the underlying relations is in one set of the partition, that there are no extraneous tuples present, and that the sets do not overlap, making P a true partition. For a query involving several aggregates, a separate partition is defined for each aggregate.

The partition for the third aggregate in the example is particularly simple.

$$P^3(x_2) = \left\{ f^{(3)} \mid \textit{Faculty}\ (f) \wedge f\ \textit{[Rank]}\ = x_2 \right\}$$

For this particular *Faculty* relation, there are two possible values for the Rank attribute: Associate and Full. P^3(Associate)= {(Merrie, Associate, 40000)} and P^3(Full) = {(Jane, Full, 44000)}. All other subsets yield empty sets, e.g., P^3(Assistant)=$\emptyset$.

The general Quel query with one aggregate has the following tuple calculus statement.

$$\text{1} \quad \left\{ w^{(j+1)} \mid (\exists t_1) \cdots (\exists t_k)(R_1(t_1) \wedge \cdots \wedge R_k(t_k) \right.$$
$$\text{2} \quad \wedge\, w[1] = d_1[a_1] \wedge \cdots \wedge w[j] = d_j[a_j] \wedge w[j+1]$$
$$= F(P(e_2[b_2], \cdots, e_l[b_l]))[b_1]$$
$$\text{3} \quad \left. \wedge\, \psi') \right\}$$

This is a simple extension of the Quel semantics without aggregates defined by Ullman [54]. This statement specifies that the result tuple w is composed of $j+1$ attributes (line one), that the tuple t_i is in the relation R_i (also line one), that the ith attribute of w is copied from the a_ith attribute of the tuple variable b_i (line two), and that the participating tuples are determined by the restriction ψ' (line three).

Line two also computes the aggregate. The appropriate partition is selected by the indicated attribute values found in the underlying relations (i.e., the values $e_2[b_2], \cdots, e_l[b_l]$). If the tuple variables appearing in the aggregate are not distinct, then the first two lines in the definition of P should be altered to eliminate duplicate tuple variables. Also, if tuple variable e_1 does not appear outside of the aggregate or in the by-clause, then that tuple variable should be removed from the first two lines of the statement just given.

The tuple calculus statement for the example is

$$\left\{ w^{(4)} \mid (\exists f)(Faculty(f) \wedge w[1] = f[Rank] \wedge \right.$$
$$w[2] = count(P^1)[Name] \wedge w[3] = count(U^2)[1] \wedge$$
$$\left. w[4] = count(P^3(f[Rank]))[Name] \right\}$$

In using tuple calculus to formalize Quel (and shortly, TQuel), we assume duplicate elimination in the resulting relation, since relations are formalized as sets.

There are six fundamental operators that perform aggregation in Quel. The grouping and selection of tuples to be aggregated is done by the partition, which also determines whether the standard or the unique version is being used. The semantics for aggregates in the outer where clause, for arbitrarily nested aggregation, and for expressions in aggregates is given elsewhere [48]. While only the semantics for the retrieve statement has been given, it is easy to extend it to specify aggregates in the Quel modification statements (**append, delete**, and **replace**) [47].

V. Tuple Calculus Semantics of TQuel Aggregates

It is convenient to base the semantics of TQuel on the conventional (snapshot) relational database model, especially because of the available mathematical foundation supporting the latter [8]. Thus the semantics of the augmented operations are expressed using traditional tuple calculus notation.

We first review the transformation of the time-specific constructs of TQuel into the tuple calculus, and briefly give the semantics of the TQuel retrieve statement, which is needed in order to introduce the semantics of temporal aggregates. This review is a condensation of [47]. The semantics of the TQuel aggregates is then developed, for the Quel analogues followed by the new TQuel aggregates.

A. Review of TQuel Semantics

As stated in the overview of TQuel in Section II, TQuel augments Quel by adding a valid clause to specify the validity time(s) of tuples, a when clause to specify the relative time ordering of the participating tuples, and an as-of clause to specify rollback in time.

The semantics makes use of several auxiliary functions: temporal constructor functions that take one or two intervals and compute an interval, and temporal predicate functions (including *overlap*) that take two intervals and compute a boolean value. All of them are ultimately defined in terms of the predicates *Before* and *Equal* and two functions *first* and *last*.

The temporal predicate τ in the when clause, containing the **precede, overlap, and, or,** and **not** operations, is transformed into a standard tuple calculus predicate Γ_τ containing only the *Before, Equal* $\wedge, \vee$, and $\neg$ operations. The valid clause is transformed into the functions Φ_ν and Φ_χ each evaluating to an event, and containing the functions *first* and *last*. The as-of clause is in fact a special when clause stating that the transaction times of the underlying tuples must overlap the (constant) interval specified in the as-of clause. The constants Φ_α and Φ_β represent the endpoints of this interval from the expressions α and β. As a consequence, the query

range of t_1 **is** R_1
...
range of t_k **is** R_k
retrieve $(d_1.a_1, \cdots, d_j.a_j)$
  valid from ν **to** χ
  where ψ
  when τ
  as of α **through** β

is translated into the tuple calculus statement

$$\Big\{ w^{(j+4)} \mid (\exists t_1) \cdots (\exists t_k)$$
$$(R_1(t_1) \wedge \cdots \wedge R_k(t_k)$$
$$\wedge\, w[1] = d_1[a_1] \wedge \cdots \wedge w[j] = d_j[a_j]$$
$$\wedge\, w[j+1] = \Phi_\nu \wedge w[j+2] = \Phi_\chi$$
$$\mathit{Before}\ (w[j+1], w[j+2])$$
$$\wedge\, w[j+3] = \textit{current transaction time}$$
$$\wedge\, w[j+4] = \infty$$
$$\wedge\, \psi'$$
$$\wedge\, \Gamma_\tau$$
$$\wedge\, (\forall i)(1 \le i \le k)(\ \mathit{overlap}\ ([\Phi_\alpha, \Phi_\beta),$$
$$[t_i[\mathit{start}], t_i[\mathit{stop}])))) \Big\}.$$

The superscript indicates that the tuple w has j explicit attributes and 4 implicit attributes, indicating an interval relation. The semantics for an event relation is similar, but with only 3 implicit attributes, since the *to* time is not present.

B. The Constant Interval Set

As we have seen, aggregates change their values over time. This will be reflected as different values of an aggregate being associated with different valid times, even in queries that look similar to Quel queries with scalar aggregates, in which no inner when or as-of clauses exist. In TQuel, the role of the external or outer *where, when* and *as of* clauses will be similar to that of the outer *where* in Quel: they determine which tuples from the underlying relations participate in the remainder of the query. These selected tuples are combined with the tuples computed from the aggregation sets to obtain the final output relation.

Aggregates always generate temporary interval relations, even though an aggregated attribute can appear in an event relation. This temporary relation has exactly one value at any point in time (for an aggregate function, the interval relation has at most one value at any point in time for each value of attributes in the by list). It is convenient to determine the points at which the value changes. Let us first define the *transition event set* of a set of relations, $R_1, \cdots, R_k$, relative to a given window function, w to be defined shortly, as

$$T(R_1, \ldots, R_k, w) \triangleq \Big\{ 0, \infty \Big\} \cup \Big\{ s \mid (\exists x)(\exists i)$$
$$(1 \le i \le k \wedge R_i(x) \wedge (s = x[\mathit{from}] \vee s = x[\mathit{to}]$$
$$\vee (\exists t)(s = t \wedge t - w(t) = x[\mathit{to}]$$
$$\wedge \forall t', t' > t, t' - w(t') > x[\mathit{to}])) \Big\}.$$

The transition event set brings together all the times when the aggregate's value could change. These times include the beginning time of each tuple, the time following the ending time of each tuple, and the time when a tuple no longer falls into an aggregation window.

The window function w is specified in the for clause. maps each time into its aggregation window size. **for each instant** implies $\forall t, w(t) = 0$; **for ever** implies $\forall t, w(t) = \infty$; and **for each** < span> implies a window size dependent on the time-stamp granularity. In the examples, an underlying granularity of month has been used. Hence, **for each month** is equivalent to **for each instant** ($\forall t, w(t) = 1-1 = 0$); **for each quarter** implies ($\forall t, w(t) = 3-1 = 2$); and **for each decade** implies ($\forall t, w(t) = 120-1 = 119$) subtracted because the window is inclusive. In all of these cases, the window function yields the same value for any input. If, however, a granularity of day is used, **for each month**, **for each quarter**, and **for each decade** would require non-constant window functions. For example, **for each month** would require w(January 31, 1980) = 31 − 1 = 30, w(February 28, 1980) = 28 − 1 = 27, and w (March 20, 1980) = 28 − 1 = 27 (since February 21, 1980, the first day in the aggregate window, was 27 days before March 20). The last line of the definition of T is somewhat complex because the aggregation window must be defined in terms of t, *not* in terms of $r[to]$. If $r[to]$ was February 21, 1980, then T would contain March 20, 1980 if **for each month** were specified. However, if $r[to]$ was February 28, 1980, then T would only contain March 31, 1980, even though March 28–30 all satisfy $t - w(t) = r[to]$.

If two times y and z are neighbors (i.e., y and z are in $T(R_1, \ldots, R_k, w)$, and no intervening time is in T), then the time interval from y to z did not witness any change in the set of relations, or in other words, all the relations remained "constant." Define then the *Constant* interval set as

$$Constant(R_1, \ldots, R_k, w)$$

$$\triangleq \Big\{ [y,z) | T(R_1, \ldots, R_k, w)(y) \wedge T(R_1, \ldots, R_k, w)(z) \wedge$$

$$y \neq z \wedge \mathit{Before}\,(y,z) \wedge (\forall e)(T(R_1, \ldots, R_k, w)(e)$$

$$\Rightarrow \mathit{Before}(e,y) \vee \mathit{Equal}(e,y) \vee \mathit{Before}(z,e) \vee \mathit{Equal}(z,e)) \Big\}.$$

The last two lines state that there is no event in the time between y and z. The constant interval set allows us to treat each constant time interval (y, z) separately, thus reducing the inner query to a number of queries, each dealing with a constant time interval. Hence, we will be able to follow the same steps as in the snapshot Quel case. For each time interval $[y, z)$ in the constant interval set a value of the aggregate, valid from y to z, will be computed and will potentially go into the result. This value is guaranteed to be unique and unchanging by the definition of *Constant.*

C. Aggregates in the Target List

For a multirelational query with one aggregate in the target list, we will take the approach used in the Quel semantics: tuples from the aggregate operation will be computed first via a partition. Initially, let F be any of the aggregate operators also defined in Quel. Consider the TQuel query with one aggregate function in the target list,

```
range of $t_1$ is $R_1$
...
range of $t_k$ is $R_k$
retrieve ($d_1.a_1, \ldots, d_j.a_j$,
   y = F($e_1.b_1$ by $e_2.b_2, \ldots, e_l.b_l$
            for $\omega$
            where $\Psi_1$
            when $\tau_1$
            as of $\alpha_1$ through $\beta_1$))
      valid from $\nu$ to $\chi$
      where $\Psi$
      when $\tau$
      as of $\alpha$ through $\beta$
```

As with Quel, the where predicate should refer only to the tuple variable e_1 or the tuple variables appearing in the by clause. The same restriction holds for the when clause appearing in the aggregate: no tuple variables are permitted in the as-of clause.

Here, the partition will be based upon the four clauses that modify the aggregate (the by, where, when, and as-of clauses). Using the same notation employed in the TQuel semantics, we may define the partition as shown at the bottom of the next page, where y and z are valid times, with *Before*(y, z) and $p = \left(\sum_{i=1}^{l} \deg(R_{e_i})\right) + 4$ (p includes the implicit attributes of e_1 only). This definition assumes that the tuple variables $e_1, \ldots, e_l$ are distinct. If they are not, then the duplicate tuple variables should be removed. In comparing this with the Quel partition, notice first that this partition is indexed both by the by-clause attribute values and by a time interval $[y, z)$. Also note that three additional lines appear here. Line 6 translates the when clause, similarly to the where clause line in the semantics of the Quel retrieve statement. Line 7 translates the as-of clause, specifying that the transaction times of all tuples of the inner query, including those in the inner where and when clauses, must overlap the rollback time specified in the as-of clause. This is similar to the as-of line in the outer query in TQuel, which will be shown shortly. The window function ω' corresponds to the keyword found in the retrieve statement. Line 8 indicates that all tuples participating in the aggregate must overlap the interval $[y, z)$. From the definition of the *Constant* interval set, which supplies the intervals $[y, z)$, it is not difficult to see that the overlapping is total. This way, aggregates will always be computed from the tuples that were valid during that interval. In determining the overlap, the window function ω' is used in a similar fashion to the definition of the transition event set. In particular, if **for ever** is specified, then ω' is the constant function returning ∞ if, and every appropriate tuple valid before time z will appear in P, yielding a cumulative aggregate. If R_{e_i} in line 8 is an event relation, the predicate should be

$$overlap([y,z), (e_i[at], e_i[at] + \omega'(y)))$$

The output relation from a query with a single aggregate in the target list is

$$\begin{array}{ll}
1 & \Big\{ w^{((j+1)+4)} | (\exists t_1) \cdots (\exists t_k)(\exists y)(\exists z) \\
2 & (R_1(t_1) \wedge \cdots \wedge R_k(t_k) \wedge \\
 & [y,z) \in \mathit{Constant}\ (R_{e_l}, \ldots, R_{e_1}, \omega') \\
3 & \wedge (\forall i)(1 \leq i \leq j)(\ \mathit{overlap}\ ([y,z), [e_i[\mathit{from}], e_i[\mathit{to}]))) \\
4 & \wedge w[1] = d_1[a_1] \wedge \cdots \wedge w[j] = d_j[a_j] \\
5 & \wedge w[j+1] = F(P(e_2[b_2], \ldots, e_l[b_l], y, z))[b_1] \\
6 & \wedge w[j+2] = \ \mathit{last}\ (y, \Phi_\nu) \wedge \\
 & w = \ \mathit{first}\ (z, \Phi_\chi) \wedge \ \mathit{Before}\ (w[j+2], w[j+3]) \\
7 & \wedge w[j+4] = \ \mathit{current\ transaction\ time} \wedge w[j+5] = \infty \\
8 & \wedge \Psi' \\
9 & \wedge \Gamma_\tau \\
10 & \wedge (\forall i)(1 \leq i \leq k) \\
11 & (\mathit{overlap}([\Phi_\alpha, \Phi_\beta), [t_i[\ \mathit{start}], t_i[\ \mathit{stop}\ ]))) \Big\}
\end{array}$$

A comparison with the tuple calculus expression for the TQuel retrieve statement given earlier reveals that lines three and five are new and lines one, two and six are altered. In line 2, the *Constant* interval set provides the interval $[y, z)$ during which the tuples are constant. It involves the relations appearing in the aggregate; the relation whose attribute is being aggregated plus all the different relations in the by-list; other relations cannot affect the aggregate. Again, these relations are assumed to be distinct for notational convenience. The window function ω' appears explicitly as an argument to the *Constant* interval set and implicitly in P. Line three ensures that the tuple variables aggregated over and those specified in the by-clause overlap with the interval during which the aggregate is constant. Line five computes the aggregate. Note that the same aggregate operator F as in the Quel semantics is used; what is different are the two additional parameters to P, y, and z, which restrict the tuples in that partition. Line six ensures that the valid time of the result relation is the intersection with the specified valid time and the interval $[y, z)$.

Two slight modifications as required for special cases. If the **valid at** ν variant is used, line 6 should be replaced with

$$w[r+2] = \Phi_\nu \wedge \ \mathit{overlap}\ ([y,z), [w[r+2], w[r+2]+1))$$

Secondly, as with the Quel semantics, if $e_1$does not appear outside of the aggregate or in the by-clause, it should also not appear in lines 1 and 2 (it *will* appear in the definition of the *Constant* interval set). Also, tuple variables mentioned in the aggregate that do not appear outside the aggregate should not appear in line 3. Unique aggregation is handled in a manner analogous to Quel's semantics.

Let us translate the original example into the tuple calculus.

```
range of f is Faculty
retrieve(f.Rank, NumFaculty=count(f.Name),
  NumRanks=countU(f.Rank),
  NumInRank=count(f.Name by f.Rank))
```

We first define a partition for each aggregate.

$$\begin{aligned}
P^1(y,z) &\triangleq \Big\{ f^{(3+2)} |\ \mathit{Faculty}\ (f) \wedge \\
&\qquad \mathit{overlap}\ ([y,z), [f[\mathit{from}\ ], f[\ \mathit{to}\ ] + 0)) \Big\} \\
P^2(y,z) &\triangleq \Big\{ f^{(3+2)} |\ \mathit{Faculty}\ (f) \wedge \\
&\qquad \mathit{overlap}\ ([y,z), [f[\mathit{from}], f[\mathit{to}] + 0)) \Big\} \\
U^2(y,z) &\triangleq \Big\{ u^{(1)} | (\exists w)(w \in P^2(y,z) \wedge u[1] = w[2]) \Big\} \\
P^3(x,y,z) &\triangleq \Big\{ f^{(3+2)} |\ \mathit{Faculty}\ (f) \wedge f[\ \mathit{Rank}\ ] = x^2 \\
&\qquad \wedge \ \mathit{overlap}\ ([y,z), [f[\mathit{from}], f[\mathit{to}] + 0)) \Big\}
\end{aligned}$$

A window size of 0 is used because the default is **for each instant**. Some instances of the values of the third partition are

$$\begin{aligned}
&P^3(\text{Assistant}, 9-71, 9-75) \\
&\quad = \{(\text{ Jane, Assistant, } 25000,\ 9-71, 12-76)\} \\
&P^3(\text{Assistant}, 9-75, 12-76) \\
&\quad = \{(\text{Jane, Assistant, } 25000,\ 9-71, 12-76), \\
&\qquad\qquad (\text{Tom, Assistant, } 23000,\ 9-75, 12-80)\}.
\end{aligned}$$

The output relation is

$$\begin{aligned}
&\Big\{ w^{(4+2)} | (\exists f)(\exists y)(\exists z)(\ \mathit{Faculty}\ (f) \wedge [y,z) \in \mathit{Constant} \\
&\qquad (\ \mathit{Faculty}\ , 0) \wedge \ \mathit{overlap}([y,z), [f[\mathit{from}], f[\mathit{to}])) \\
&\qquad \wedge\ w[1] = f[\mathit{Rank}] \\
&\qquad \wedge\ w[2] = \mathit{count}\,(P^1(y,z))[\mathit{Name}] \\
&\qquad \wedge\ w[3] = \mathit{count}\,(U^2(y,z))[1]
\end{aligned}$$

$$\begin{array}{ll}
1 & P(x_2, \ldots, x_l, y, z) \triangleq \{ u^{(p)} | (\exists) e_1) \cdots (\exists e_l) \\
2 & \qquad (R_{e_1}(e_1) X \cdots X R_{e_l}(e_l) \\
3 & \qquad \wedge u = e_1 || \cdots || e_l \\
4 & \qquad \wedge e_2[b_2] = x_2 X \cdots X e_l[b_l] = x_l \\
5 & \qquad \wedge \Psi'_1 \\
6 & \qquad \wedge \Gamma_{\tau_l} \\
7 & \qquad \wedge (\forall i)(1 \leq i \leq l)\ \mathit{overlap}\ ([\Phi_{\alpha_1}, \Phi_{\beta_1}, [e_i[\ \mathit{start}\ ], e_i[\ \mathit{stop}\ ])) \\
8 & \qquad \wedge (\forall i)(1 \leq i \leq l)\ \mathit{overlap}\ ([y,z), [e_i[\ \mathit{from}\ ], e_i[\mathit{to}] + \omega'(y))) \\
9 & \qquad)\}
\end{array}$$

$$\begin{aligned}
&\wedge\ w[4] = count\,(P^3(f[Rank], y, z))[Name]\\
&\wedge\ w[5] = last\,(y, f[from])\\
&\wedge\ w[6] = first\,(z, f[to]) \wedge Before\,(w[5], w[6])\\
&\quad \wedge\ overlap\,([f[from], f[to]), [now, now+1))\\
&\quad)\}
\end{aligned}$$

The last two lines correspond to the default valid and when clauses. Since the underlying relations are historical, the lines involving transaction time are not necessary.

The semantics of unique aggregation, of multiple aggregation, of aggregates in the outer where, when, and valid clauses, of aggregates with no by-clause, and of arbitrarily nested aggregation in TQuel is given elsewhere [48].

D. Operators for the New TQuel Aggregates

Let us specify the semantics of the new aggregates introduced in Section II-C by specifying their aggregate operators. As discussed above, the aggregate operators, e.g., *count*, for the Quel aggregates, e.g., `count`, which are also permitted in TQuel, are identical to their Quel counterparts.

Let R be an event relation of degree j (the degree only concerns the explicit attributes) with n tuples, $n > 2$, and let t be a tuple variable associated with R. Since R is an event relation, it contains an implicit valid-at time-stamp attribute, denoted at. All except $rising_i$ compute a single snapshot tuple of degree j. We first define a function that induces a total ordering on the tuples in a relation.

Definition:

$$\begin{aligned}
S \triangleq chronorder\,(R)\\
&\Leftrightarrow (\forall i)(1 \le i \le |S|)((\exists t)(R(t) \wedge t = S_i))\\
&\wedge Before(S_{i-1}[at], S_i[at])\\
&\wedge S_{i-1}[at] \ne S_i[at])
\end{aligned}$$

where $|S|$ is the length of the sequence S, and S_i is the i^{th} element of S. Each element of S is a full tuple from R, and the elements of S are ordered by the at times of R. If several tuples in R show identical at times, only one of them is taken into S. Hence, the length of S is less than or equal to n. We use the *Before* predicate rather than "<" to later accommodate indeterminacy.

Definition:

$$rate(R) \triangleq \left[\left[\frac{1}{|S|-1}\sum_{i=1}^{|S|-1}\frac{S_{i+1}[1]-S_i[1]}{S_{i+1}[at]-S_i[at]}\right],\cdots,\right.$$
$$\left.\left[\frac{1}{|S|-1}\sum_{i=1}^{|S|-1}\frac{S_{i+1}[j]-S_i[j]}{S_{i+1}[at]-S_i[at]}\right]\right]$$

where $S = chronorder\,(R)$ and $|S| > 1$. Each attribute of the result tuple equals the average increment (positive or negative) in the values of the corresponding attribute in R, per unit of time (the default is the time-stamp granularity, defined in Section II). An optional per clause can be used to specify the span desired; this causes multiplication of the result by a fixed conversion factor. For example, if time-stamp granularity was a millisecond and the user specified "**`per month`**" then the computed result is multiplied by the conversion factor of milliseconds to months (2.592×10^9) before being output.

Definition:

$$var\,(R) \triangleq sd\,(G(R))/mean\,(G(R))$$

where $G(R) \triangleq < g_1, \cdots, g_{|S|-1} >$ (i.e., the ordered sequence of durations of the tuples in R), such that $S = chronorder\,(R), |S| > 1$, implies that $(\exists i)(1 \le i \le |S| - 1 \wedge g_i = S_{i+1}[at] - S_i[at])$, and $mean\,(X)$ and $sd\,(X)$ respectively denote the arithmetic mean and the arithmetic standard deviation of the real numbers in the set X. Each attribute of the result tuple equals the variability of the spacing between the at times among the tuples in R. This is in fact the coefficient of variation of the set $G(R)$. Note that var returns a single value, rather than a tuple.

Observe that $mean\,(D(R))$ is never zero since $S_i[at]$ and $S_{i+1}[at]$ are distinct. Not necessarily all tuples from R will make their way into S; S was so defined in order to ensure that $rate$ or var will not attempt a division by zero. Should the user need to specify which of the tuples from T has to be chosen for the chronological order, one of the other aggregates can be used to create a temporary relation R that contains the relevant tuples, and then `rate` or `var` may be applied to T.

Definition: $firstagg\,(R) \triangleq t_{first}$, where t_{first}satisfies the predicate

$$R(t_{first}) \wedge (\forall t)(R(t) \wedge t \ne t_{first} \Rightarrow \textit{Before}\ (t_{first}[at], t[at])$$
$$\vee \textit{Equal}(t_{first}[at], t[at])).$$

The resulting tuple is the tuple whose valid times contain the earliest time of a tuple in R, more specifically, no other tuple in R began before t_{first}. If R is empty, $t_{first} = (0, \ldots, 0, 0, \infty)$. The *firstagg* function supports the `first` aggregate.

Definition: $earliest\,(R) \triangleq t_{first}[at]$,where t_{first} satisfies the predicate given above. The result is the event represented by the earliest tuple in the relation. *lastagg* and *latest*, and *earliest* over intervals, have analogous definitions [48].

The last function supports the `rising` aggregate over the attribute with the index i.

Definition: $rising_i\,(R) \triangleq [earliest\,(\,maximal_i\,(R)), latest\,(R))$, where $maximal_i\,(R)$ satisfies the predicate

$$\begin{aligned}
&latest(R) \in maximal_i(R)\\
&\wedge \forall t_1, t_2 \in maximal\,(R)(Before(t_1, t_2) \Rightarrow t_1[i] \le t_2[i])\\
&\wedge \exists t_1 \in maximal_i(R) \exists t_2 \in R\\
&\neg \exists t_3 \in R(Before\,t_2[at], t_3[at]) \wedge Before(t_3, t_1)\\
&\wedge \neg\, Equal\,(t_2[at], t_3[at]) \wedge \neg\, Equal\,(t_3, t_1) \wedge t_2[i] > t_1[i]))
\end{aligned}$$

The first predicate states that the interval terminates at the last event. The second predicate states that the value is indeed rising, and the third predicate states that the interval is maximal, that is, that in an immediately preceding event the attribute fell.

In summary, aggregate operators exist for all TQuel aggregates. The semantics of aggregates appearing in all possible positions within the retrieve statement has been specified. This semantics is easily extendible to the append, delete, and replace statements in TQuel.

VI. Implementation Aspects

TQuel and its semantics are declarative in nature. In order to implement the language, a more operational form is required.

We have defined an algebra for historical relations [34]. In this section, we discuss how aggregates are supported in this algebra and show how TQuel statements containing aggregates may be translated into the historical algebra. We also examine how the aggregate operators in the algebra may be implemented, focusing on incremental materialization.

A. Aggregates in the Historical Algebra

The *historical relational algebra*, an extension of the conventional relational algebra, supports valid time. Unlike TQuel's data model, historical relations manipulated by this algebra are attribute-value time-stamped, though it is a simple matter to convert between that representation and tuple time-stamping. The historical relational algebra contains historical versions of the projection, selection, union, difference and cartesian product operators: $\hat{\pi}, \hat{\sigma}, \hat{\cup}, \hat{-}$ and $\hat{\times}$. A new operator, *historical derivation* ($\hat{\delta}$), that performs a combination of historical selection and projection, is also available.

We also defined two new operators, $\hat{A}$ and $A\hat{U}$, that perform aggregation over historical operators. The aggregate operator is denoted by $\hat{A}_{f,w,N,X,N'}(Q,R)$; its unique variant is $\hat{AU}_{f,w,N,X,N'}(Q,R)$. R and Q are historical relations. N is the attribute (in R's schema) on which the aggregate is applied. Q supplies the values that partition R and X denotes the attributes on which the partitioning is applied, with the restrictions that $Attr\,(Q) \subseteq Attr\,(R)$ and $\{N\} \cup X \subseteq Attr\,(Q)$. f is the name of the aggregate operator, e.g., *count* for the `count` aggregate.

If X is empty, the historical aggregate operators simply calculate a single distribution of scalar values over time for an arbitrary aggregate applied to attribute N of relation R. The computed value is appended to each tuple of R, and is given the name N'. The interval(s) of validity of the aggregate is recorded in that attribute's time-stamp. When X is empty, the tuples in Q are ignored.

If X is not empty, the operators calculate, for each subtuple in Q formed from the attributes X, a distribution of scalar values over time for an aggregate applied to attribute N of the subset of tuples in R whose values for attributes X match the values for the same attributes of the tuple in Q. Hence, X corresponds to the by-list. Generally $X = Attr\,(Q)$ and $Q = \pi_X(R)$, but these constraints are not dictated by the formal definition of $\hat{A}$.

Let us translate the original example into the algebra.

```
range of f is faculty
retrieve (f.Rank,
    NumFaculty = count (f.Name),
    Numranks = countU(f.Rank),
    NumInRank = count(f.name by f.Rank))
```

$$
\begin{aligned}
&\hat{\pi}_{\text{Rank, NumFaculty, NumRanks, NumInRank}}(\\
&\quad \hat{\delta}_{\Gamma \text{Name} \cap \text{NumFaculty} \cap \text{NumRanks} \cap \text{NumInRank}}(\\
&\qquad \hat{\sigma}_{\text{Rank} = \text{Agg.Rank}}(\\
&\qquad\quad Faculty\\
&\qquad\quad \times \hat{A}_{\text{count}, 0, \text{Name}, \emptyset, \text{NumFaculty}}(Faculty, \emptyset)\\
&\qquad\quad \hat{\times} A\hat{U}_{\text{count}, 0, \text{Rank}, \emptyset, \text{NumRanks}}(Faculty, \emptyset),\\
&\qquad\quad \hat{\times} \hat{A}_{\text{count}, 0, \text{Name}, \{\text{Rank}\}, \text{NumInRank}}(\\
&\qquad\quad Faculty,\ \hat{\pi}_{\text{Rank}}(Faculty)))))
\end{aligned}
$$

where Γ is a temporal predicate equivalent to the default when clause, and $\hat{\delta}$ performs temporal selection and projection, and ensures that each resulting tuple has identical time-stamps for all attributes (allowing conversion back into a tuple time-stamped representation). For all three aggregates we used a constant window function of 0, corresponding to **`for each instant`** (the default). For the first two aggregates, which contain no by clause, the fourth subscript to the aggregate operator is an empty set, as is the second parameter. The third aggregate does have a by clause, so we project out those attributes from the *Faculty* relation to provide the second parameter to the aggregate operator, and also link the body of the retrieve statement with this aggregate through the selection predicate.

Elsewhere we give the tuple calculus semantics of the $\hat{A}$ and $A\hat{U}$ operators, as well as the algebraic equivalents of the TQuel retrieve statement with aggregates in its target list, in its where, when, and valid clauses, and in the where and when clauses within another aggregate, and argue that this method of converting TQuel aggregates to their algebraic equivalents can also handle an arbitrary level of nesting of aggregates [48]. We also prove that the tuple calculus semantics of the algebraic translation of a TQuel retrieve statement is equivalent to the tuple calculus semantics of the original statement, and argue that the same holds for TQuel retrieve statements containing an arbitrary number of aggregates. This proved the theorem that the language formed by embedding the historical algebra (which only supports valid time) in the commands used to support transaction time (given elsewhere [33]) has the expressive power of TQuel.

B. Implementing the $\hat{A}$ and $A\hat{U}$Operators

Epstein has developed an aggregate processing strategy for Quel aggregates [15]. Briefly, the strategy for each aggregate proceeds as follows.

1. If it is an aggregate function (i.e., has a by-list), then create a temporary to hold the results.
2. If the aggregate function has a qualification, project the by-list into $temp_1$, with the result attribute initialized appropriately, e.g., to 0 for `count`.
3. If the aggregate is multivariable or unique project the qualifying tuples into $temp_2$.
4. If the aggregate is unique, remove duplicates from $temp_2$.
5. Compute the aggregate by scanning $temp_2$, looking up the tuple in $temp_1$ with the same by-list values and updating the aggregate value in $temp_1$. If the aggregate is a scalar aggregate, then simply update its value as $temp_2$ is scanned.
6. If it is an aggregate function, link $temp_1$ into the outer query by equating the by-list attributes.

In the tuple calculus semantics presented in Sections IV and V, the partition $P(x_2, \cdots, x_l)$ partitioned the $temp_2$ relation. In the algebraic operators $\hat{A}$ and $A\hat{U}$, the first parameter R is $temp_2$ and the second parameter Q is $temp_1$. For multiple and nested aggregation, Epstein advocates using multiple temporary relations. The approach discussed here to support in the tuple calculus semantics aggregates in various locations in the retrieve statement, and nested aggregation, is an application of that general idea.

To extend this strategy to accommodate time-varying relations, conceptually we need a version of $temp_1$ for every time interval in which R remained constant. In fact, the tuple calculus semantics effectively does this by making the end points of the constant interval $[y, z)$ arguments to the partitioning function. The algebraic operator may do this more effectively by recording multiple intervals, with each interval associated with a single aggregate value (for aggregate functions, each combination of by-list values in the $temp_1$ relation would be associated with multiple intervals). As each tuple in $temp_2$ processed, the interval-value pairs would be updated. As an example, let's simulate the processing of the first few tuples in the *Faculty* relation for the instantaneous aggregate `count(f.Rank for each instant)`. This is a scalar aggregate, so there is only one collection of intervals, initialized to the single interval $[-\infty, \infty)$, with a value of 0. When the first tuple, (Jane, Assistant, 25000, 9–71, 12–76), is processed, we divide the single interval into three: $([-\infty, 9-71), 0), ([9-71, 12-76), 1), ([12-76, \infty), 0)$. The second tuple, (Tom, Assistant, 23000, 9–75, 12–80), overlaps two of these intervals, and thus results in one additional interval.

$$\begin{aligned}&([-\infty, 9-71), 0), ([9-71, 9-75), 1),\\&\quad([9-75, 12-76), 2), ([12-76, 12-80), 1),\\&\quad([12-80, \infty), 0)\end{aligned}$$

This process continues for each tuple of $temp_2$. For the cumulative aggregates, the effect of each tuple extends into the future. For count(f.Rank **for each year)**, the first tuple divides the initial interval into the three intervals.

$$([-\infty, 9-71), 0), ([9-77, 12-77), 1), ([12-77, \infty), 0)$$

Note that the second interval ends at 12–77, rather than 12–76 for the instantaneous version. Unlike conventional aggregates, the space requirements of $temp_1$ are not fixed after step 2, above. However, the effects of this expansion can be ameliorated somewhat by pre-allocating storage, and by exploiting any temporal ordering or locality contraints in the underlying relations [26].

C. An Incremental Aggregate Operator

A promising approach to achieve greater efficiency in temporal DBMS's is that of incremental view materialization [6], [19], [20], [40]. This process brings the view up-to-date following the update of one of its underlying relations by identifying the tuples that must be inserted into, and the tuples that must be deleted from, the view's old state for the view's new state to be consistent with the new states of its underlying relations, without having to recompute the view itself. The net changes that an update operation makes to a stored relation, either a base relation or a materialized view, is termed the relation's *differential.*

Incremental view materialization is more efficient than processing without views if four conditions are satisfied simultaneously: 1) the number of queries against a view is sufficiently higher than the number of updates to its underlying relations, 2) the sizes of the underlying relations are sufficiently large, 3) the selectivity factor of the view predicate is sufficiently low, and 4) the percentage of the view retrieved by queries is sufficiently high. Since these conditions are rather restrictive in practice, commercial DBMS's do not support incremental view materialization.

One reaches a different conclusion when considering historical relations. The storage structure may be organized in such a way that updates are more costly than those to a conventional relation by perhaps only a constant factor. However, retrievals are more costly by a factor that is roughly sublinear to linear in the size of the relation [2], [3]. While update cost remains fairly constant, retrieval costs increases monotonically over time. At some point, probably quite soon, incremental view materialization becomes beneficial for most temporal views. An added benefit of incremental view materialization is a greatly reduced response time, which is critical in some applications, such as process control and the stock market. For example, if the query of the *stocks* relation given previously was implemented incrementally, tuples such as those shown in Table VII could be displayed as soon as the underlying data was received, in this case perhaps a few seconds after 9:20.

Hence, it is desirable that the historical algebra be able to support incremental view materialization. We have defined an alternate, incremental semantics for the historical operators, including $\hat{A}$ and $A\hat{U}$. In this semantics, each operator is defined as a mapping from one (or two) relation states and its (their) differential onto a resulting relation state and its corresponding differential [32].

$$\hat{A}^{I}_{j, w, N, X, N'}(R, \Delta_R, Q, \Delta_Q)$$

The output differential for this operator depends on an input relation's state just before an update as well as the input relation's differential for the update. Hence, both relation states and differentials are required as inputs to the incremental operators. Furthermore, because the output of one operator must acceptable as input to another operator, the output must include, for definitional purposes, its output relation's state just before an update, as well as its output relation's differential for the update. Note, however, that this requirement need not be extended to an implementation of the algebra. If an implementation were to cache, either virtually or physically, the input relations to each operator, only differentials would need to be computed and passed among operators.

Some aggregates, such as `sum` and `count`, need not cache the input relation at all; others, such as `avg`, need only cache summary information needed to compute the aggregate. Aggregates such as `min` and `earliest` may require the entire input relation to be cached. If the tuple containing the value of the `min` aggregate is deleted, then the original relation would need to be consulted to compute the new minimum. To improve the efficiency of maintaining aggregates

TABLE VII

Stock	Price	VarSpacing	GrowthRate	from	to
NRC	39	0.35	0.27	9:05	9:20

of this latter type, Hanson suggest that a queue of possibly duplicate candidate aggregate values, rather than a single value, be maintained [19]. Then, if a change to an aggregate's underlying relation changes or deletes a tuple containing the aggregate's value, the aggregate's new value could be assumed to be the next element in the queue. Only if the queue were empty would the aggregate have to be recomputed. This technique can be extended to apply to historical aggregates by maintaining queues for each constant interval. Appropriate data structures for maintaining such queues have yet to be studied.

Another optimization is to merge the processing of the aggregate operator and the (incremental) projection operator present in the second argument of the aggregate operator when a by-list is specified, e.g., computing the *NumInRank* attribute in the example presented earlier. The projection operator is not required to cache the entire input relation; instead it need retain only the number of input tuples contributing to each output tuple. Differentials containing added tuples increase this number; deleted tuples reduce this number. The number can be stored as another attribute of $temp_1$. In fact, for the instantaneous count aggregate, this number is identical to the value of the aggregate.

The historical algebra can thus support aggregates in both unmaterialized views (via query modification [49]) and materialized views, and can support various view maintenance strategies, such as in-line view evaluation, immediate-recomputed materialization, and immediate-incremental materialization in computing aggregates. In concert with techniques developed for rollback relations [25], it can also support these maintenance strategies for views defined on temporal relations that incorporate both valid and transaction time.

VII. Related Work

As was mentioned in the introduction, most conventional query languages include support for aggregates. There has also been some formal work on aggregates. Klug introduced an approach to handle aggregates within the formalism of both relational algebra and tuple relational calculus [30]. His method makes it possible to define both standard and unique aggregates in a rigorous way. Ceri and Gottlob present a translation from a subset of SQL that includes aggregates into the relational algebra, thereby defining an operational semantics for SQL aggregates [7]. Nakano's translation of SQL into the relational algebra is more comprehensive, as it includes optimization and accommodates null values [35]. Also, significant progress has been made in the area of *statistical databases* [57], [60], [61]. Such databases, used primarily for summary statistics gathering and statistical analysis, contain set-valued attributes. Klug's relational algebra and calculus have been extended to manipulate set-valued attributes and to utilize aggregate functions [56], [58], [59], thereby forming a theoretical framework for statistical database query languages.

Aggregates may also be found in several of the dozen query languages supporting time that have appeared over the last decade. In some of these languages, aggregates play only a small role. Ben-Zvi included several aggregate operators and functions in his TRM language, although not in a comprehensive manner [5]; Ariav also mentioned aggregates in the context of his TOSQL language [4]. Although Gadia's HTQuel language (an extension of Quel) does not explicitly include aggregates (there is no way to perform an aggregate such as **count** over an explicit attribute in HTQuel), his "temporal navigation" operators (e.g., First) can be simulated using aggregated temporal constructors in TQuel, since they effectively extract an interval from a collection of intervals [17]. The Lambda query language, another extension of SQL, also includes aggregates [1]. Instantaneous aggregates are made available in the Time Relational Algebra by permitting SQL queries, which can incorporate aggregates, to be used as arguments to algebraic operators [31].

Finally, five other languages supporting time include a comprehensive set of aggregates and associated constructs. Legol 2.0 was one of the first time-oriented query languages to appear [27]. This language is based on the relational algebra. HQuel, an extension of Quel, is based on a model incorporating set-valued, time-stamped attributes [51]. It is supported by an algebra that includes an enumeration operator useful for aggregation [52], [53]. TSQL [36] and HSQL [43] are extensions of SQL [24] incorporating valid time. The operations over the time sequence collections of the temporal data model (*TDM*), presented in an SQL-like syntax, include AGGREGATE and ACCUMULATE statements [45]. The Temporal Extended Entity Relationship (TEER) model and associated query language was subsequently proposed [14].

A detailed evaluation of aggregates in Quel, TQuel, Legol, HQuel, TSQL and TDM against a set of nineteen criteria is presented elsewhere [48]. TQuel satisfies all but one criterion: an implementation does not yet exist for TQuel aggregates. An early version of Legol has been implemented, but it is not stated whether aggregates were implemented in this prototype; Quel aggregates have been implemented; no other proposal that supports time has been implemented. None of the other languages have a formal semantics. Without such a formal definition, the meaning of each construct, and the interaction between constructs, is unclear. Instantaneous aggregates were introduced by Jones; only Legol, TEER and TQuel support such aggregates. Moving window aggregates and temporal partitioning were introduced by Navathe and Ahmed in TSQL; only TQuel and TEER, and perhaps TDM and HSQL, also support these aspects. Tansel introduced the concept of an average weighted by the duration of the values [51]; TQuel's rate aggregate serves a similar purpose. Tansel also introduced the concept of a *proportional sum* adjusted by the duration of validity of the value; this adjustment can be performed in TQuel by using a (non-aggregated) duration function. The other languages do not provide such aggregates.

VIII. Summary

This paper makes four contributions. First, a formal semantics for the conventional query language Quel was presented. The simple case of aggregates in the target list was considered in detail; the remaining cases of aggregates in the outer where clause, arbitrarily nested aggregation, and expressions in aggregates are given elsewhere [48]. This completes the formal definition of Quel (the core of the retrieve statement and the modification statements were previously formalized in [54] and [47], respectively).

Secondly, the aggregates in Quel have been extended in a minimal fashion for inclusion in TQuel. All Quel aggregates are permitted in TQuel. TQuel added the when and as-of clauses, which are the temporal analogues for valid and transaction time, respectively, to the where clause. These clauses are permitted within the aggregate. The for clause was added to distinguish between instantaneous, cumulative, and moving window aggregates. Several additional temporal aggregate operators were also introduced. The resulting language subsumes all aspects of aggregates appearing in other proposals, including temporal partitioning and an average weighted by duration.

Third, the Quel tuple calculus semantics was extended to accommodate time-varying relations. Our approach used the *Constant* interval set and a transition event set to determine those intervals over which a relation remains static, enabling a time-varying aggregate value to be computed. Again, only the simple case of aggregates in the target list was considered, though we did accommodate by, for, where, when, and as of clauses within the aggregate. The semantics of unique aggregation, of multiple aggregation, of aggregates in the outer where, when, and valid clauses, of aggregates with no by-clause, and of arbitrarily nested aggregation in TQuel is given elsewhere [48]. This semantics preserves snapshot reducibility, making a Quel aggregate behave identically whether evaluated on a snaphot or a temporal database. The semantics also is independent of the time-stamp granularity. The result is a complete formal semantics for TQuel and its snapshot subset Quel. A complete formal semantics for no other relational query language, conventional or temporal, has been defined.

Finally, a temporal relational algebra has also been defined that fully supports TQuel and its aggregates [32], [33], [34], thus providing an consistent operational semantics for the language. We examined how the batch and incremental aggregate operators in the algebra could be implemented.

More work is required in developing efficient implementations. In particular, data structures to store the constant intervals, and to store the queues required for some incremental processing techniques, need to be developed. Extensions to existing query optimization strategies to handle aggregates need to be investigated; proposals for optimization of conventional aggregates [11], [16], [23], [28], [29], [39] provide a good place to start. Similarly, previous work on processing aggregates with hard time constraints [22] should be applied both to the batch and to the incremental evaluators described above.

Because the semantics is expressed in terms of the constant interval set, this semantics can be easily extended to handle aggregates on possibilistic data (c.f., [41], [42]). The semantics could also be extended to handle SQL-type null values by adapting Nakano's rule-based translation method [35] or the Extended Three Valued Predicate Calculus [37]. Accommodating *historical indeterminacy*, where the exact time that an even occurred is not known [12], appears to be more challenging.

Finally, aggregates over transaction time (c.f., [44]) and user-defined temporal aggregates (c.f., [18], [55]) should also be investigated.

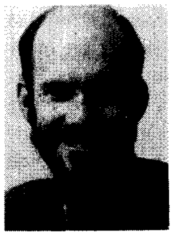
Acknowledgment

We are grateful to Peter Bloomfield for his remarks on the requirements of experimental data in statistical time series that lead to the creation of the `var` operator, to Ravi Krishnamurthy for identifying the need for aggregates that follow rising and falling stock market prices, and to Ilsoo Ahn, David Beard, Juan Valiente and the reviewers for helpful comments on this paper.

References

[1] M. E. Adiba and N. Bui Quang, "Historical multi-media databases," in *Proc. Conf. Very Large Databases*, Y. Kambayashi, Ed., Aug. 1986, pp. 63–70.
[2] I. Ahn and R. Snodgrass, "Performance evaluation of a temporal database management system," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, C. Zaniolo, Ed., Association for Computing Machinery, May 1986, pp. 96–107.
[3] ——, " Performance analysis of temporal queries," *Information Sciences*, vol. 49, pp. 103–146, 1989.
[4] G. Ariav, "A temporally oriented data model," *ACM Trans. Database Syst.*, vol. 11, pp. 499–527, Dec. 1986.
[5] J. Ben-Zvi, "The time relational model," PhD. dissertation, Computer Science Department, UCLA, 1982.
[6] J. A. Blakeley, P.-A. Larson, and F. W. Tompa, "Efficiently updating materialized views," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, C. Zaniolo, Ed., Association for Computing Machinery, pp. 61–71, May 1986.
[7] S. Ceri and G. Gottlob, "Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 324–345, Apr. 1985.
[8] E. F. Codd, "Relational completeness of data base sublanguages," in *Data Base Systems*, vol. 6 of Courant Computer Symposia Series. Englewood Cliffs, N. J.: Prentice Hall, 1972, pp. 65–98.
[9] C. J. Date, *A Guide to INGRES*. Reading, MA: Addison-Wesley, 1987.
[10] C. J. Date, *An Introduction to Database Systems*, vol. 1, Fifth Edition of Systems Programming Series. Reading, MA: Addison-Wesley, 1990.
[11] U. Dayal, "Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers," in *Proc. Conf. Very Large Databases*, P. Hammersley, Ed., Sept. 1987, pp. 197–208.
[12] C. E. Dyreson and R. T. Snodgrass, "Valid-Time indeterminacy," in *Proc. 9th Int. Conf. Data Engineering,* pp. 335–343, Apr. 1993.
[13] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*. Benjamin/Cummings, 1989.
[14] R. Elmasri and G. Wuu, "A temporal model and query language for ER databases," in *Proc. 6th Int. Conf. Data Engineering*, pp. 76–83, Feb. 1990.
[15] R. Epstein, "Techniques for processing of aggregates in relational database systems," *UCB/ERL M7918*, Computer Science Department, University of California, Berkeley, Feb. 1979.
[16] J. C. Freytag and N. Goodman, "Translating aggregate queries into iterative programs," in *Proc. Conf. Very Large Databases*, Y. Kambayaski, Ed., Aug. 1986, pp. 138–146.
[17] S. K. Gadia and J. H. Vaishnav, "A query language for a homogeneous temporal database," in *Proc. ACM Symp. Principles of Database Systems*, Mar. 1985, pp. 51–56.
[18] E. N. Hanson, "User-defined aggregates in the relational database system INGRES," Master's thesis, Computer Science Department, University of California, Berkeley, Dec. 1984.
[19] ——, "Efficient support for rules and derived objects in relational database systems," PhD. dissertation, Computer Science Department, University of California, Berkeley, Aug. 1987.
[20] ——, "A performance analysis of view materialization strategies," in *Proc. ACM SIGMOD Annual Conf.*, U. Dayal and I. Traiger, Eds., Association for Computing Machinery, ACM Press, May 1987, pp. 440–453.
[21] G. D. Held, M. Stonebraker and E. Wong, "INGRES—A relational data base management system," in *Proc. AFIPS National Computer Conf.*, May 1975, pp. 409–416.
[22] W.-C. Hou and G. Özsoyoğlu, and B. K. Taneja, "Processing aggregate relational queries with hard time constraints," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, May 1989, pp. 68–77.

[23] ——, "Statistical estimators for aggregate relation algebra queries," *ACM Trans. Database Syst.*, vol. 16, pp. 600–654, Dec. 1991.
[24] IBM, "SQL/Data-system, concepts and facilities," *Rep. GH24-5013-0*, Jan. 1981.
[25] C. S. Jensen, L. Mark, and N. Roussopoulos, "Incremental implementation model for relational databases with transaction time," *IEEE Trans. Knowledge Data Eng.*, vol. 3, pp. 461–473, Dec. 1991.
[26] C. S. Jensen and R. Snodgrass, "Temporal specialization and generalization," *IEEE Trans. Knowledge Data Eng.*, to be published.
[27] S. Jones, P. Mason, and R. Stamper, "LEGOL 2.0: A relational specification language for complex rules," *Information Systems*, vol. 4, pp. 293–305, Nov. 1979.
[28] W. Kiessling, "SQL-like and Quel-like correlation queries with aggregates revisited," *Tech. Rep. UCB/ERL Memo 84/75*, Electronics Research Laboratory. Sept. 1984.
[29] ——, "On semantic reefs and efficient processing of correlation queries with aggregates," in *Proc. Conf. Very Large Databases*, A. Pirotte and Y. Vassiliou, Eds. Stockholm, Sweden: Aug. 1985, pp. 241–250.
[30] A. Klug, "Equivalence of relational algebra and relational calculus query languages having aggregate functions," *J. of the Association of Computing Machinery*, vol. 29, pp. 699–717, July 1982.
[31] N. A. Lorentzos and R. G. Johnson, "An extension of the relational model to support generic intervals," in *Extending Data Base Technology 88*. Venice, Italy: Mar. 1988.
[32] E. McKenzie, "An algebraic language for query and update of temporal databases," PhD. dissertation, Computer Science Department, University of North Carolina at Chapel Hill, Sept. 1988.
[33] E. McKenzie and R. Snodgrass, "Schema evolution and the relational algebra," *Information Systems*, vol. 15, pp. 207–232, June 1990.
[34] ——, "Supporting valid time in an historical relational algebra: Proofs and extensions, *Tech. Rep. TR 91-15*, Department of Computer Science, University of Arizona, Aug. 1991.
[35] R. Nakano, "Translation with optimization from relational calculus to relational algebra having aggregate functions," *ACM Trans. Database Syst.*, vol. 15, pp. 518–557, Dec. 1990.
[36] S. B. Navathe and R. Ahmed, "A temporal relational model and a query language," *Information Sciences*, vol. 49, pp. 147–175, 1989.
[37] M. Negri, S. Pelagatti, and L. Sbattella, "Formal semantics of SQL queries," *ACM Trans. Database Syst.*, vol. 16, pp. 513-534, Sept. 1991.
[38] P.E. O'Neil, "Revisiting DBMS benchmarks," *Datamation*, vol. 35, pp. 47–54, Sept. 1989.
[39] A. Rosenthal and D. Reiner, "Extending the algebraic framework of query processing to handle outerjoins," in *Proc. Tenth Int. Conf. on Very Large Data Bases*, Aug. 1984.
[40] N. Roussopoulos, "An incremental access method for ViewCache: Concept, algorithms, and cost analysis," *ACM Trans. Database Syst.*, vol. 16, pp. 535-563, Sept. 1991.
[41] E. A. Rundensteiner and L. Bic, "Aggregates in possibilistic databases," in *Proc. Conf. Very Large Databases*, 1989.
[42] ——, "Evaluating aggregates in possibilistic relational databases," *Data and Knowledge Engineering*, vol. 7, pp. 239-267, 1992.
[43] N. Sarda, "Extensions to SQL for historical databases," *IEEE Trans. Knowledge Data Eng.*, vol. 2, pp. 220–230, June 1990.
[44] E. Sciore, "Using annotations to support multiple kinds of versioning in an object-oriented database system," *ACM Trans. Database Syst.*, vol. 16, pp. 417-438, Sept. 1991.
[45] A. Segev and A. Shoshani, "Logical modeling of temporal data," in *Proc. ACM SIGMOD Annual Conf. Management of Data*, U. Dayal and I. Traiger, Eds., Association for Computing Machinery, ACM Press, May 1987, pp. 454–466.
[46] R. Snodgrass and I. Ahn, "Temporal databases," *IEEE Computer*, vol. 19, pp. 35–42, Sept. 1986.
[47] R. Snodgrass, "The temporal query language TQuel," *ACM Trans. Database Syst.*, vol. 12, pp. 247–298, June 1987.
[48] R. Snodgrass, S. Gomez, and E. McKenzie, "Aggregates in the temporal query language TQuel," *Tech. Rep. TR–89–26*, Department of Computer Science, University of Arizona, Nov. 1989.
[49] M. Stonebraker, "Implementation of integrity constraints and views by query modification," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, Association for Computing Machinery, June 1975.
[50] M. Stonebraker, E. Wong, P. Kreps, and G. Held, "The design and implementation of INGRES," *ACM Trans. Database Syst.*, vol. 1, pp. 189–222, Sept. 1976.
[51] A. U. Tansel and M. E. Arkun, "HQuel, A query language for historical relational databases," in *Proc. Third Int. Workshop on Statistical and Scientific Databases*, July 1986.
[52] A. U. Tansel and M. E. Arkun, "Aggregation operations in historical relational databases," in *Proc. Third Int. Workshop on Statistical and Scientific Databases*, July 1986.
[53] A. U. Tansel, "A statistical interface for historical relational databases," in *Proc. Int. Conf.Data Engineering*, IEEE Computer Society Press, Feb. 1987, pp. 538–546.
[54] J. D. Ullman, *Principles of Database and Knowledge-Base Systems*. Potomac, MD: Computer Science Press, 1988, vol. 1.
[55] S. Wensel, "The POSTGRES reference manual," *Tech. Rep. UCB/ERL M88/20*, University of California, Mar. 1988.
[56] G. Özsoyoğlu and Z. M. Özsoyoğlu, "Statistical database query languages," *IEEE Trans. Software Eng.*, 1990.
[57] G. Özsoyoğlu, Z. M. Özsoyoğlu, and V. Matus, "Extending relational algebra and relational calculus with set-valved attributes and aggregate functions," *ACM Trans. Database Syst.*, vol. 12, pp. 566–592, Dec. 1987.
[58] ——, "An extension of relational algebra for summary tables," in *Proc. Second Int. Workshop on SDB Management*, Sept. 1983, pp. 202–212.
[59] ——, "Summary-Table-By-Example: A database query language for manipulating summary data," in *Proc. COMPDEC Conf.*, Nov. 1984.
[60] *Proc. First Int. Workshop on Statistical Database Management*, H. K. Wong, Ed., 1981.
[61] *Proc. Second Int. Workshop on Statistical Database Management*, J. McCarthy, Ed., 1983.

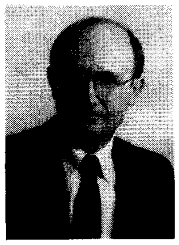


Richard Snodgrass received the B.A. degree in physics from Carleton College in 1977 and the Ph.D. degree in computer science from Carnegie Mellon University in 1982.

Until 1989 he was on the faculty at the University of North Carolina. He is now an Associate Professor in the Department of Computer Science at the University of Arizona. His research interests include temporal databases, programming environments, and persistent object stores. He directed the design and implementation of the Scorpion meta-environment. He is the author of the book *The Interface Definition Language: Definition and Use* (Computer Science Press). He is an Associate Editor of ACM TODS and will chair the program committee for *SIGMOD* '94.

Santiago L. Gomez received the M.S. degree in computer science from the University of North Carolina in 1988.

He is now at the Centro Nacional de Computacion, Asuncion, Paraguay.



L. Edwin McKenzie received the B.S. in mathematics from Clemson University in 1971, the M.S. degree in systems engineering from the Air Force Institute of Technology, Dayton, OH in 1976, and the Ph.D. degree in computer science from the University of North Carolina in 1988.

He is currently the Director of Systems Technology, Air Force Computer Acquisitions Center, Air Force Communications Command, Hanscom AFB, MA. His research interests are in computer performance evaluation and temporal databases.