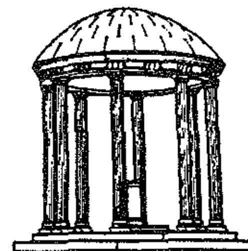
Aggregates in the Temporal Query Language TQuel

TR86-009

March 1986

Richard Snodgrass and Santiago Gomez

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175



UNC is an Equal Opportunity/Affirmative Action Institution.

TR 86-009

Aggregates in the

Temporal Query Language TQuel

March, 1986

Richard Snodgrass & Santiago Gomez

Department of Computer Science
University of North Carolina
Chapel Hill, North Carolina 27514

Abstract

This paper defines aggregates in the temporal query language TQuel and provides 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, valid, and as-of clauses; nested aggregation; and instantaneous, cumulative, and unique variants are supported. These aggregates give the user a rich set of statistical functions that range over time, while requiring minimal additions to TQuel and its semantics.

This work was supported by NSF grant DCR-8402339 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.

Table of Contents

1. Introduction 1
    1.1. Aggregates in Conventional Query Languages 1
    1.2. Aggregates in Time-Oriented Databases 2
    1.3. Structure of the Paper 3

2. Aggregates In Quel 5
    2.1. Informal Specification of Quel Aggregates 5
    2.2. Semantics of the Quel Retrieve Statement 7
    2.3. Adding Aggregates to Tuple Relational Calculus 8
    2.4. Unique Aggregation 12
    2.5. Aggregates in the Outer Where Clause 13
    2.6. Nested Aggregation 13
    2.7. Expressions in Aggregates 14
    2.8. Summary 14

3. Temporal Aggregates in TQuel 16
    3.1. Adding Aggregates to TQuel 19
    3.2. Cumulative versus Instantaneous Aggregates 21
    3.3. New Aggregates 23
    3.4. Some Examples 25
    3.5. Defaults 29
    3.6. Syntax Summary 29

4. Tuple Calculus Semantics of TQuel Aggregates 31
    4.1. Review of TQuel Semantics 31
    4.2. New TQuel Aggregates 33
    4.3. The Constant Predicate 35
    4.4. Instantaneous Aggregates 37
    4.5. Cumulative Aggregates 41
    4.6. Mixing Different Aggregates in a Query 45
    4.7. Aggregates in the Outer Where Clause 45
    4.8. Nested Aggregation 46
    4.9. Unique Aggregation 47
    4.10. Aggregates in the Other Outer Clauses 48

5. Conclusion 49
Acknowledgements 50

References 51

Table of Figures

Figure 1: Example Relations shown on a Time Line 18

Figure 2: An Example of count 20

Figure 3: Instantaneous versus Cumulative versus Unique Aggregates 22

Figure 4: Converting an Event Relation into an Interval Relation 42

Table of Examples

Example 1: How many faculty members are there in each rank? 6

Example 2: How many faculty members and different ranks are there? 7

Example 3: What was Jane's rank when Merrie was promoted to Associate? 17

Example 4: How many faculty members were there each time a paper was submitted to a journal? 25

Example 5: Who was making the second smallest salary, and how much was it, during each period of time prior to 1980? 25

Example 6: Who were the professors hired into or promoted to a rank while the first faculty member ever in that rank was still in that category? 26

Example 7: How many different salary amounts has the department paid its members since its creation until 1981? 26

Example 8: For each faculty member, list his/her last rank and first journal they submitted a paper to. 27

Example 9: Given the above set of experimental data, how equally spaced are the observations in time? 28

Chapter 1

INTRODUCTION

A database represents objects in the real world, their relationships, and the ways they combine with each other. The effective modeling of reality has often been difficult within conventional databases. Conventional databases are really snapshot databases, since they show the state of the real world at one particular point in time; they do not support the passage of time. Several researchers have proposed extensions to database models and query languages that include time as an intrinsic part.

This paper deals with the formal semantics of aggregates in a temporal relational database query language. In this chapter, we will first see how aggregates are handled in a conventional query language. Then we will turn our attention to aggregates in time-oriented relational databases.

1.1. Aggregates in Conventional Query Languages

A convenient way of combining data from tuples in a relational database is through aggregate operators. They include such common statistics as the count, the sum, and the average of the values of an attribute in a given relation (or part of a relation). Most commercially available relational database management systems (DBMS's) provide several aggregate operations [Date 1983, SQL/DS 1981, Ullman 1982]. One of the best known systems among these is Ingres [Stonebraker et al. 1976]. Quel [Held et al. 1975], the query language used with Ingres, is fairly comprehensive in its support of aggregates. Techniques for implementing non-temporal aggregates in a relational query language are discussed by Epstein, with a description of the method used by Quel for processing queries that involve arbitrarily complex aggregations of data [Epstein 1979]. However, no formalization of the semantics of aggregates in Quel has been done to date.

Klug introduced an approach to handle aggregates within the formalism of both relational algebra and tuple relational calculus [Klug 1982]. 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 relational algebra, thereby defining an operational semantics for SQL aggregates [Ceri & Gottlob 1985]. Klug's approach was exploited in this translation; his approach will be used later in this paper to build a formal semantics for aggregates in Quel.

Finally, significant progress has been made in the area of *statistical databases* [LBL 1981, LBL 1983]. 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 [Ozsoyoglu, et al. 1986], thereby forming a theoretical framework for statistical database query languages. As such languages manipulate non-first-normal-form relations, they are of limited relevance in this paper.

1.2. Aggregates in Time-Oriented Databases

A variety of work has been done on incorporating time in a database (c.f., [Anderson 1982, Bradley 1978, Bubenko 1977, Clifford & Warren 1983, Codd 1979, Lum et al. 1984, Sernadas 1980, Snodgrass & Ahn 1986]). The majority of the researchers focus on issues such as how to model time, how to treat time attributes in a manner consistent with the way humans view time, and how to represent temporal information in an efficient manner.

Few researchers, however, have investigated aggregates in time-oriented relational databases. One of the first time-oriented query languages, the Legol 2.0 language, included aggregates [Jones & Schwan 1979, Jones & Mason 1980]. In Legol 2.0, all the instances of aggregation are performed over time because every tuple is time stamped. A distinction was made between aggregates at each time, or *instantaneous aggregates*, that yield a distribution on the time axis, and aggregates over all time, or *cumulative aggregates*, that yield a single value:

Instantaneous	*Cumulative*
max	highest
min	lowest
sum	accumulate
number	count

Furthermore, the purely temporal aggregates "first", returning the earliest time, and "last", returning the latest time, were defined. No formal semantics was provided.

Ben-Zvi included several aggregate operators and functions in his TRM language, although not in a very clear or comprehensive manner [Ben-Zvi 1982]; Ariav also mentioned aggregates in the context of his TOSQL language [Ariav 1985]. Finally, although Gadia's HTQuel language does not explicitly include aggregates, his "temporal navigation" operators (e.g., First) can be simulated using aggregates, since they effectively extract an interval from a collection of intervals [Gadia & Vaishnav 1985].

Aggregates in TQuel, a superset of Quel incorporating temporal constructs, were previously defined informally [Snodgrass 1982]. Both instantaneous and cumulative versions of aggregates were given. The difficulties caused by tuples that are duplicated over time and indeterminacy of the temporal attributes, were identified and several alternative definitions of cumulative aggregates were provided. As with the previous attempts, no satisfactory semantics was given.

1.3. Structure of the Paper

For temporal relational database systems, some work has been done on aggregates, but they still need a formal definition. It is necessary to specify the meaning of each aggregate from a user's point of view and to specify a rigorous semantics for each of them. Formal definitions are also useful for implementing the query language.

We will begin by constructing a formal semantics of aggregates in Quel. Then, an intuitive introduction to temporal aggregates in TQuel will be given in Chapter 3. Chapter 4 is devoted to developing a formal semantics for aggregates in TQuel.

Throughout the paper, a fixed-width font is used for functions and operators in the query language (e.g., `count`), and italics is used for functions in the semantics (e.g., *count*).

Chapter 2

AGGREGATES IN QUEL

This chapter will present a complete semantics for the Quel aggregates, as a convenient point of reference for the TQuel semantics to be developed in Chapter 4. An informal specification for aggregates is given, followed by a formal semantics of the retrieve statement with aggregates in the Quel language.

2.1. Informal Specification of Quel Aggregates

The Quel operations for aggregation are

count The number of values that exist for a given attribute in a relation. Since every attribute has exactly one value in each tuple, this operator yields the same result on all attributes of a relation.

any An indicator of whether there exists at least one tuple in a relation. It returns a 1 if the relation is non-empty and 0 otherwise.

sum The sum of the values present for a given attribute. This operator can be computed only on a numeric attribute.

avg The average, or arithmetic mean, of the values present for a given attribute. The average is defined in the usual way, i.e. the sum divided by the count. Because of this dependency upon sum, the **avg** is also a numeric-attribute-only operator.

min The smallest of the values present for a given attribute. For an alphanumeric attribute, the alphabetical ordering is used to determine the smallest element.

max The largest of the values present for a given attribute. For an alphanumeric attribute, the alphabetical ordering is used to determine the largest element.

These operators can be used in two types of aggregation:

(a) *Scalar aggregates*, yielding a *single value* as the result.

(b) *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.

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.)

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.

EXAMPLE. Suppose the relation *Faculty* holds relevant data, say name, rank and salary, about the professors in a university department:

Faculty(Name, Rank, Salary):

Name	*Rank*	*Salary*
Tom	Assistant	23000
Merrie	Assistant	25000
Jane	Associate	33000

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

Example 1: How many faculty members are there in each rank?

The range statement declares a tuple variable f that will be associated to *Faculty* throughout the query. The retrieve statement contains the target list of attributes to be derived for the output relation, in this case, *Rank* and *NumInRank:*

Rank	*NumInRank*
Assistant	2
Associate	1

The output relation contains as many tuples as actual values exist in the by-list. If there had been no by-list, *NumInRank* would be 3 in all the derived tuples. ∎

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.

EXAMPLE. This example illustrates multiple aggregates and unique aggregation.

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

Example 2: How many faculty members and different ranks are there?

The result is a single tuple:

NumFaculty	*NumRanks*
3	2

∎

2.2. Semantics of the Quel Retrieve Statement

A tuple relational calculus semantics for Quel statements without aggregates was defined by Ullman [Ullman 1982] and will be reviewed here. Although attribute values in a target list can be expressions in general, we ignore that detail in this paper for simplicity of notation. Thus the skeletal Quel statement is

range of t_1 **is** R_1
....
range of t_k **is** R_k
retrieve $(t_{i_1}.D_{j_1}, \ldots, t_{i_r}.D_{j_r})$
  where ψ

in which

$$1 \le i_1 \le k, \ldots, 1 \le i_r \le k$$
$$1 \le j_1 \le deg(R_{i_1}), \ldots, 1 \le j_r \le deg(R_{i_r})$$

$deg(R)$ is the ***degree*** of R, that is, the number of attributes in each tuple of R. The corresponding tuple calculus statement is

$$\Big\{ w^{(r)} \mid (\exists t_1) \cdots (\exists t_k)$$
$$(R_1(t_1) \wedge \cdots \wedge R_k(t_k)$$
$$\wedge\ w[1] = t_{i_1}[j_1] \wedge \cdots \wedge w[r] = t_{i_r}[j_r]$$

$$\wedge \psi') \}$$

This statement specifies that the tuple t_i is in the relation R_i, the result tuple u is composed of r attributes, the m-th attribute of u is copied from the j_m -th attribute of the tuple variable t_{i_m}, and that the participating tuples are determined by the restriction ψ'. We use ψ' instead of ψ to indicate modifications for attribute names and Quel syntax conventions.

2.3. Adding Aggregates to Tuple Relational Calculus

The semantics for the Quel retrieve statement with aggregates will be presented now. We first introduce the *aggregate operators* to be used in the tuple calculus. This material is new, and is based on Klug's method [Klug 1982].

Let R be a relation of degree r containing n tuples, $n \geq 0$, and let t be a tuple variable associated with R.

DEFINITION. $count(R) \triangleq (n, ..., n)$

That is, the count operator yields a tuple whose r components equal n.

DEFINITION. $any(R) \triangleq (sign(n), ..., sign(n))$

The sign function produces the value +1 if n is positive (at least one tuple in R), and 0 if n is zero (no tuples in R). Again, all r components of the result tuple equal the same value.

For the remaining definitions, assume $n > 0$.

DEFINITION. $sum(R) \triangleq \left(\sum_{t \in R(t)} t[1], ..., \sum_{t \in R(t)} t[r] \right)$

Each component of the result tuple equals the sum of all values in the corresponding component of the tuples of R.

DEFINITION. $avg(R) \triangleq \left(\frac{1}{n} \sum_{t \in R(t)} t[1], ..., \frac{1}{n} \sum_{t \in R(t)} t[r] \right)$

Each component of the result tuple equals the average or arithmetic mean of all values in the corresponding component of the tuples of R.

DEFINITION. $min(R) \triangleq (\min_{t \in R(t)} t[1], ..., \min_{t \in R(t)} t[r])$

Each component of the result tuple equals the minimum of all values in the corresponding component of the tuples of R.

DEFINITION. $max(R) \triangleq (\max_{t \in R(t)} t[1], ..., \max_{t \in R(t)} t[r])$

Each component of the result tuple equals the maximum of all values in the corresponding component of the tuples of R.

For $n = 0$, *sum*, *avg*, *min* and *max* are arbitrarily defined to be 0. However, new implementations can be more consistent with reality if they return a special null value for those cases.

The advantage of defining aggregate operators to work on relations instead of on domains is that duplicate values enter the set calculations without difficulty. Later on we consider unique aggregates which eliminate duplicate values to compute aggregates over unique values.

The functions are used in the tuple calculus semantics. Let F be any of the aggregates defined in Section 2.1. Quel queries with one aggregate function are of the form

range of t_1 **is** R_1
. . .
range of t_k **is** R_k
retrieve $(t_{i_1}.D_{j_1}, ..., t_{i_r}.D_{j_r}, y = \mathrm{F}(t_{l_1}.D_{m_1}$ **by** $t_{l_2}.D_{m_2}, ..., t_{l_n}.D_{m_n}$ **where** $\psi_1))$
where ψ

in which

$$1 \le i_1 \le k, ..., 1 \le l_r \le k$$
$$1 \le l_1 \le k, ..., 1 \le l_n \le k$$
$$1 \le j_1 \le deg(R_{i_1}), ..., 1 \le j_r \le deg(R_{i_r})$$
$$1 \le m_1 \le deg(R_{l_1}), ..., 1 \le m_n \le deg(R_{l_n}).$$

Again, we simplify the expressions appearing in the aggregate to attribute names. There is also the restriction that the tuple variable(s) mentioned in ψ_1 must be either t_{l_1} or one of the tuple variables appearing in the by clause: $t_{l_2}, ..., t_{l_n}$. The attributes outside the aggregate, $D_{j_1}, \cdots, D_{j_r}$, and the attributes used within the aggregate, $D_{m_2}, \cdots, D_{m_n}$, usually overlap, but need not. This aggregate

(a) takes the cartesian product of the relations associated with the tuple variables appearing in the aggregate,

(b) removes all resulting tuples that do not satisfy the condition in the where clause of the aggregate,

(c) partitions the resulting tuples by the values of the attributes listed in the by clause,

(d) applies the aggregate to each partition,

(e) and finally associates the result with each combination of tuples participating in the original query, with the partition selected using the values indicated in the by clause.

We first specify the partition of the cartesian product of the relations associated with the tuple variables appearing in the aggregate. Initially assume that the tuple variables t_{i_1}, ..., t_{i_n} are all distinct. Define a *partitioning function* P corresponding to the aggregate in the query as a function of $n-1$ values a_2, ..., a_n, given by

$$P(a_2, ..., a_n) \triangleq \Big\{ t^{(p)} \mid (\exists t_{i_1}) \cdots (\exists t_{i_n}) \\ (R_{i_1}(t_{i_1}) \wedge \cdots \wedge R_{i_n}(t_{i_n}) \\ \wedge t = t_{i_1} \\ \wedge t_{i_2}[m_2] = a_2 \wedge \cdots \wedge t_{i_n}[m_n] = a_n \\ \wedge \psi_1') \Big\}$$

where $p \triangleq deg(R_{i_1})$. Each of the combinations of values a_2, ..., a_n existing in the specified attributes produces one partition on which the aggregate has to be applied.

EXAMPLE. The partitioning function for Example 1 is particularly simple:

$$P(a_2) = \Big\{ t^{(2)} \mid (\exists f)(Faculty(f) \wedge t = f \wedge f[rank] = a_2) \Big\}$$

For this particular *Faculty* relation, P(Assistant) = {(Tom, Assistant, 23000), (Merrie, Assistant, 25000)} and P(Associate) = {(Jane, Associate, 33000)}. Note that we use attribute names rather than indices for notational convenience. ∎

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 r-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 $f(P(a_2, ..., a_n))[m]$ denote the m-th attribute of the tuple evaluated by $f(P(a_2, ..., a_n))$. For Example 1, $count(P(\text{Assistant})) = \{(2, 2, 2)\}$ and $count(P(\text{Assistant}))[Name] = 2$.

The counterpart tuple calculus statement for the Quel query is then

$$\Big\{ w^{(r)} \mid (\exists t_1) \cdots (\exists t_k)$$
$$(R_1(t_1) \wedge \cdots \wedge R_k(t_k)$$
$$\wedge\ w[1] = t_{i_1}[j_1] \wedge \cdots \wedge\ w[r] = t_{i_r}[j_r]$$
$$\wedge\ w[r+1] = f(P(t_{l_2}[m_2], ..., t_{l_n}[m_n]))[m_1]$$
$$\wedge\ \psi') \Big\}$$

The partitioning function computes the partitions, with the appropriate partition selected by the parameter passed to P. 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.

EXAMPLE. The tuple calculus statement for Example 1 is

$$\Big\{ w^{(2)} \mid (\exists f)(Faculty(f) \wedge w[1] = f[rank] \wedge w[2] = count(P(f[Rank]))[Name]) \Big\} \ ∎$$

For a scalar aggregate, there is no by clause and the partitioning function P is simpler, namely

$$P \triangleq \Big\{ t^{(p)} \mid (\exists t_{l_1})(R_{l_1}(t_{l_1}) \wedge t = t_{l_1} \wedge \psi_1') \Big\}$$

Here, P is formulated to emphasize its similarity with the more general partitioning function given earlier. As expected, P computes a subset of R_{l_1}. The tuple calculus statement for the query remains the same as above, except that P is used in place of $P\ (t_{l_2}[m_2], ..., t_{l_n}[m_n])$.

EXAMPLE. For the count aggregate of Example 2,

$$P_1 = \left\{ t^{(2)} \mid (\exists f)(Faculty(f) \wedge t = f\ ) \right\} \blacksquare$$

For a query involving several aggregates $f_1, ..., f_k$, a separate partitioning function P (of either the scalar or the function form) is defined for each aggregate.

2.4. Unique Aggregation

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 partitioning function P solves the problem.

Let the modified partitioning function be defined in terms of P as

$$U\ (a_2, ..., a_n) \triangleq \left\{ w^{(1)} \mid (\exists b)(b \in P(a_2, ..., a_n) \wedge w[1] = b[m_1]) \right\}$$

The net effect of this is the elimination of all duplicate values from the attribute upon which aggregation will be performed.

For a scalar unique aggregate, the partitioning function U of degree p (rather than $p+q$) is defined in a similar fashion based on P,

$$U \triangleq \left\{ w^{(1)} \mid (\exists b)(b \in P \wedge w[1] = b[m_1]) \right\}$$

EXAMPLE. For Example 2 for the countU aggregate,

$$P_2 = \left\{ t^{(2)} \mid (\exists f)(Faculty(f) \wedge t = f\ ) \right\}$$

$$U_2 = \left\{ u^{(1)} \mid (\exists b)(b \in P_2 \wedge u[1] = b[1]) \right\}$$

$$= \{ (\text{Assistant}), (\text{Associate}) \}$$

The final tuple calculus expression is then

$$\left\{ w^{(2)} \mid (\exists f)(Faculty(f) \wedge w[1] = count(P_1)[Name] \wedge u[2] = count(U_2)[Rank]) \right\} \blacksquare$$

The tuple calculus semantics of all 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*.

2.5. Aggregates in the Outer Where Clause

So far we have seen standard and unique aggregates being used in the target list of a query. They can also appear in an expression in the Quel where clause, alone or with other terms like constants and values from attributes.

Let us first deal with an aggregate in the main where clause. If it is a scalar aggregate, it is independent of the rest of the query and therefore it is simply calculated and replaced by its value. However, if an aggregate function appears in the outer where clause, its corresponding partitioning function is defined, and the values of the aggregated attribute are used in place of the aggregate in the query. Following the rule that the tuple variables in by-lists are global, the by clause is linked to the rest of the query through the arguments to the partitioning function.

2.6. Nested Aggregation

A similar rule applies in the case of nested aggregation, that is, when an aggregate function f_1 appears in a local where clause of an aggregate f_2. The tuple variables in the by-list of f_1 are linked to the tuple variables of the same name appearing in their outer environment (that is, the f_2 query).

Nesting may be deeper, with f_2 nested in (called from) an outer aggregate f_3. Again, if tuple variables with the same name appear in the by-list of f_2 and in the f_3 query, they will be linked, and so on. Links are accomplished via the arguments to the partitioning functions. Thus, at any one time, only one level of nesting need be considered [Epstein 1979].

2.7. Expressions in Aggregates

In the formal semantics, we assumed that a single attribute was aggregated, after partitioning by zero or more attribute values. Quel allows arbitrary expressions to be aggregated, and supports expressions in the by clause. The former can be accommodated by simply substituting appropriate expressions for attributes in the line relating the output attribute in the main tuple calculus statement.

EXAMPLE. If Example 1 was modified to

```
range of f is Faculty
retrieve (f.Rank, This = count(f.Name by f.Rank) * count(f.Salary by f.Rank))
```

the only change would be in the computation of $w[2]$:

$$w[2] = count(P(f[Rank]))[Name] * count(P(f[Rank]))[Salary] \quad \blacksquare$$

Expressions in the by clause require two changes: one in the definition of the partitioning function where the parameters are equated and one in the main statement, where values of the parameters are specified.

EXAMPLE. If Example 1 was modified to

```
range of f is Faculty
retrieve (f.Rank, This = count(f.Name by f.Salary mod 1000))
```

the modified partioning function definition and tuple calculus statement would be

$$P(a_2) = \left\{ t^{(2)} \mid (\exists f)(Faculty(f) \wedge t = f \wedge f[Salary]\ mod\ 1000 = a_2) \right\}$$

$$\left\{ w^{(r)} \mid (\exists f)(Faculty(f) \wedge w[1] = f[Rank] \wedge w[2] = count(P(f[Salary]\ mod\ 1000))[Name]) \right\} \quad \blacksquare$$

2.8. Summary

There are six fundamental operators that perform aggregation in Quel. The grouping and selection of tuples to be aggregated is done by the partitioning function, which also determines whether the standard or the unique version is being used. Aggregates may appear in the outer

where clause, as well as nested in the inner where clause. The depth of nesting can be arbitrary.

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**) [Snodgrass 1986], using the strategy discussed in this chapter. It is also straightforward to extend the aggregates to operate over arbitrary expressions.

Now that the tuple calculus semantics of Quel with aggregates is complete, we can use these results for defining a tuple calculus semantics for TQuel aggregates.

Chapter 3

TEMPORAL AGGREGATES IN TQUEL

In Chapter 2 we have seen the various Quel aggregates and their formal semantics. In this chapter we introduce TQuel aggregates in an intuitive way through examples. We first give an overview of the TQuel language and then turn to aggregates.

TQuel is a version of Quel, augmented to handle the time dimension [Snodgrass 1986]. Relations in TQuel can represent either a collection of events that happen *at* certain points in time (event relations), or a collection of entities that have a duration, that is, a *beginning* and an *to* in time (interval relations). Thus, event relations have a distinguished valid time attribute, *at*, and interval relations have two distinguished valid time attributes, *from* and *to*.

TQuel differentiates between the valid time and the transaction time in a database [Snodgrass & Ahn 1986], in a way similar to that of Lum and Dadam's logical and physical time [Lum et al. 1984]. Both event and interval relations carry two transaction-time attributes, *start* and *stop*. The assignment of the transaction times to a target relation is made by the system when data are recorded. 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.

To simplify the exposition, we will not use transaction time, and hence the as-of-through clause, in the examples. All relations will be *historical* relations, containing only the *from* and *to* valid times.

EXAMPLE. The relations ***Faculty, Submitted*** and ***Published***, drawn from [Snodgrass 1986], are assumed to contain the following tuples:

Faculty(Name, Rank, Salary):

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

Submitted(Author, Journal):

Author	*Journal*	*at*
Jane	CACM	11-79
Merrie	CACM	9-78
Merrie	TODS	5-79
Merrie	JACM	8-82

Published(Author, Journal):

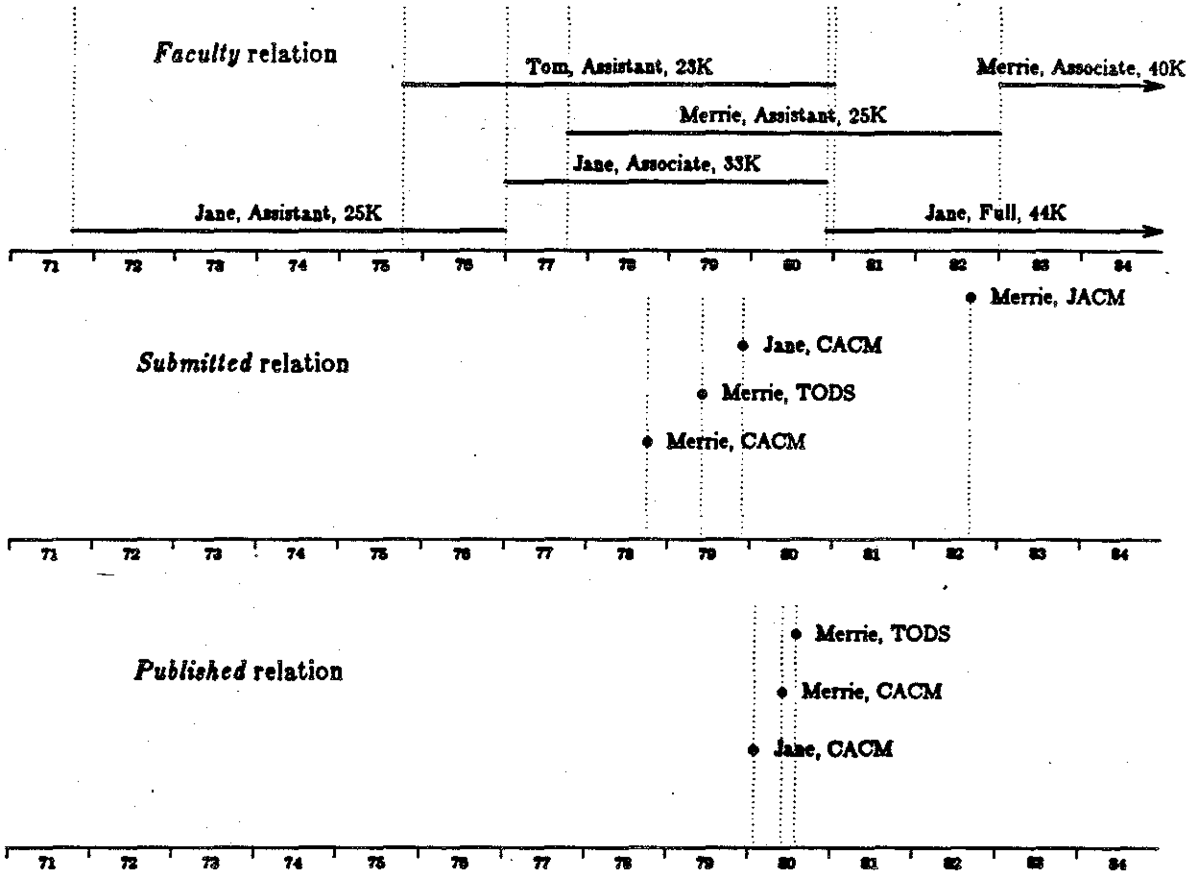
Author	*Journal*	*at*
Jane	CACM	1-80
Merrie	CACM	5-80
Merrie	TODS	7-80

A representation of the tuples in the three relations is shown in Figure 1. A faculty member's salary is assumed, for the sake of this example, to change only on promotion.

```
range of f1 is Faculty
range of f2 is Faculty
retrieve (f1.Rank)
valid at begin of f1
where f1.Name = "Jane" and f2.Name = "Merrie" and f2.Rank = "Associate"
when f1 overlap begin of f2
```

Example 8: What was Jane's rank when Merrie was promoted to Associate?

Figure 1: Example Relations shown on a Time Line



Only two tuples will participate in this query, (Jane, Full, 44000, 11-80, ∞) for `f1` and (Merrie, Associate, 40000, 12-82, ∞) for `f2`, based on the where and when clauses. The target list specifies the value of the *Rank* attribute and the valid-at clause specifies the value of the implicit *at* attribute. The resulting relation has one tuple,

Rank	*at*
Full	12-82

3.1. Adding Aggregates to TQuel

It is desirable that TQuel aggregates be a superset of the Quel aggregates, with a natural time-oriented interpretation. Therefore, the TQuel version of a Quel aggregate will perform the same fundamental operation, while ranging over an event or an interval relation.

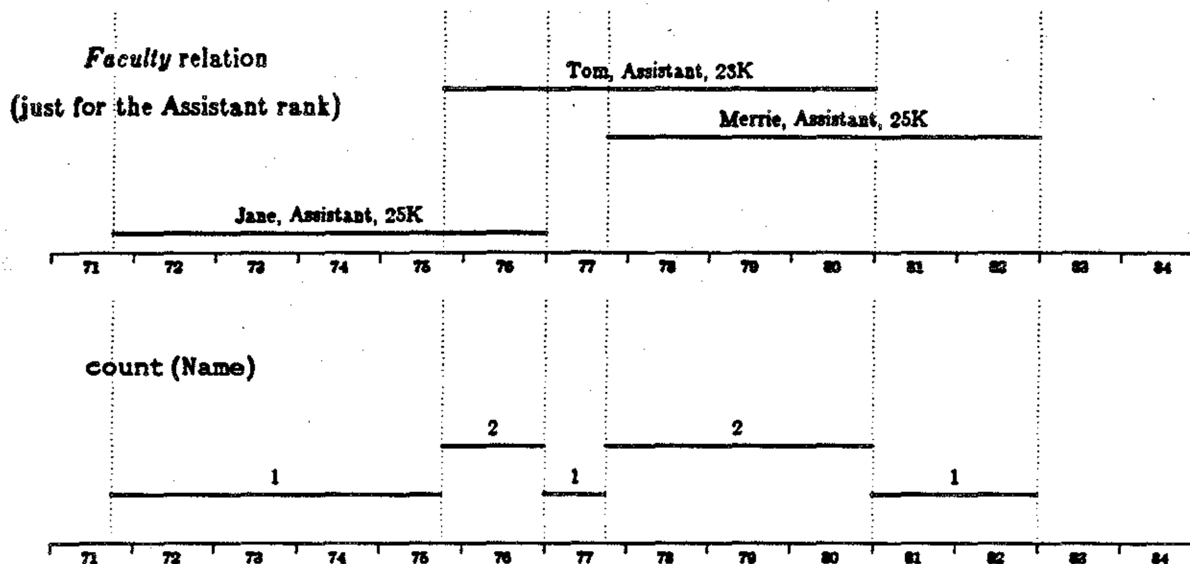
There are some differences between Quel and TQuel aggregates. Historical and temporal databases are characterized by the changing condition of their relations: at time t_1 a relation contains a set of tuples, and at time t_2 the same relation may contain a different set. Since aggregates are computed 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 attached to its valid times. For an aggregate with a by-list, a sequence of values for each value in the by-list is generated.

EXAMPLE. Let us consider Example 1, this time on an historical relation:

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

The most intuitive approach is to retrieve each tank, together with the number of faculty at that rank. As can be seen in Figure 2, for each rank there can be more than one related count.

Figure 2: An Example of count



The query yields the following tuples

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

This query, formulated as if it were a Quel query, outputs the *history* of the requested count, which is a time-varying function. When a Faculty tuple is created, or becomes invalid, the *count* changes its value. Thus each tuple output is valid between two events (represented by vertical dotted lines) in the graph of the *Faculty* relation (Figure 1). Notice that no tuples are generated with a size of zero. ▮

Therefore, the correct way to determine valid times for the output tuples is to

(a) compute the valid times from the valid clause in the query, and then
(b) create a result tuple for each interval of time when an aggregate value overlaps the interval given by those computed valid times.

3.2. Cumulative versus Instantaneous Aggregates

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

Cumulative Aggregates. If the value returned by an aggregate for each point t in time is computed from all tuples that have been valid since the beginning of the retrieval interval up to and including t, regardless of whether the tuples are still valid at t, then the aggregate is said to be *cumulative*.

Instantaneous Aggregates. If the value returned by an aggregate for each point t in time is computed only from the tuples valid at time t, then the aggregate is said to be *instantaneous*.

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 *timestamp granularity*) is reduced, the probability of finding any valid tuples decreases [Snodgrass 1982]. Aggregates such as count, applied at a given instant, would thus have different values depending upon the length of the time unit, which is not a good feature. 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 timestamp 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. This distinction may need to be reassessed when "fuzzy" events are supported (c.f., [Snodgrass 1982]). However, each value of an aggregate, be it instantaneous or cumulative, is valid during a period of time.

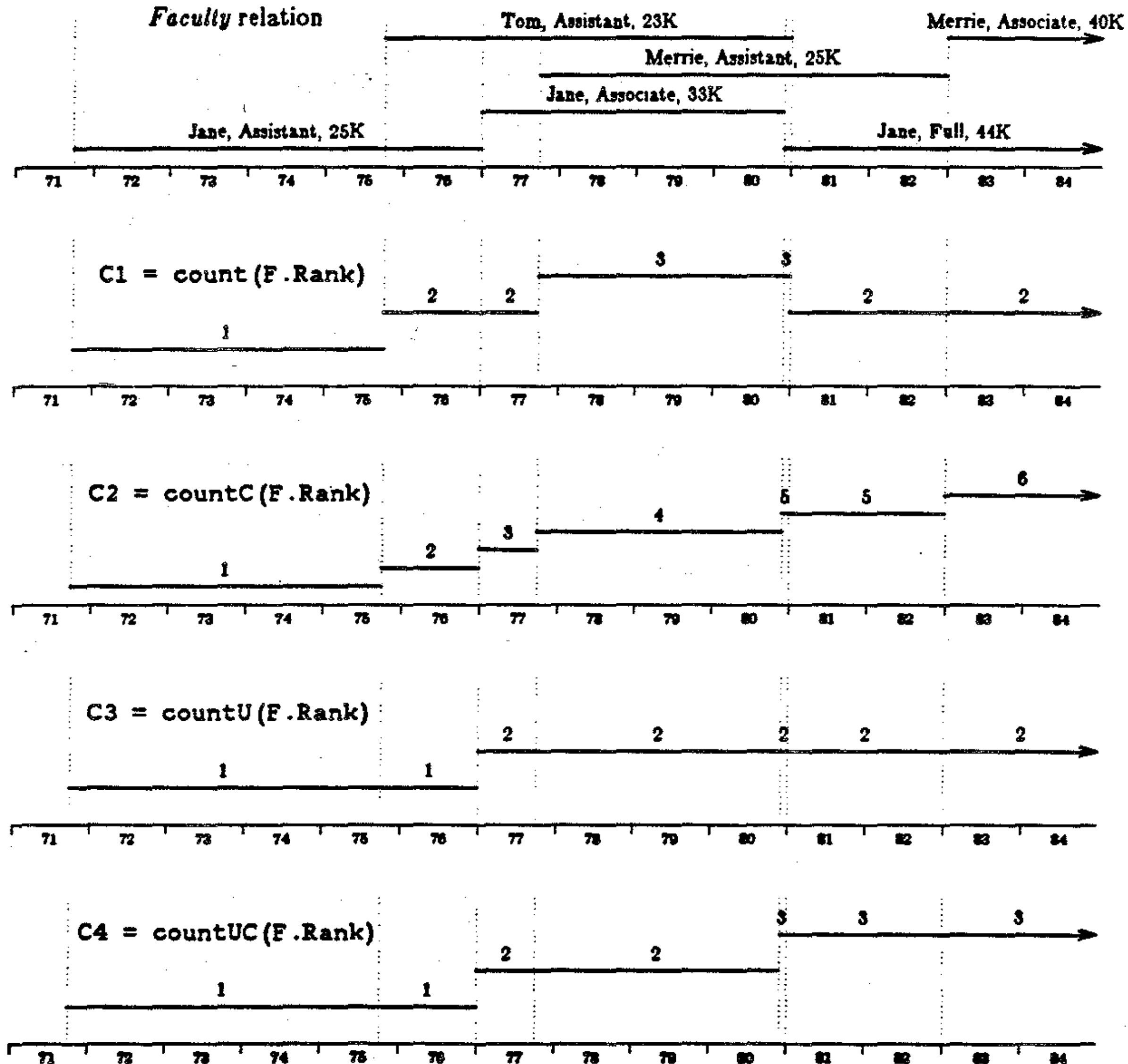
Each Quel aggregate has two TQuel versions: one for the instantaneous and the other for the cumulative case. The cumulative version (defined for both event and interval relations) will have the same name as the instantaneous version, with a "c" appended to it.

EXAMPLE. To illustrate the difference between instantaneous, cumulative, instantaneous unique and cumulative unique aggregates of an interval relation, consider the diagram of the output of an instantaneous aggregate compared to the output of the cumulative one in Figure 3, which illustrates the execution of the following query,

```
range of f is Faculty
retrieve (C1 = count(F.Rank), C2 = countC(F.Rank), C3 = countU(F.Rank),
          C4 = countUC(F.Rank))
```

on the historical *Faculty* relation shown in Figure 1. ∎

Figure 3: Instantaneous versus Cumulative versus Unique Aggregates



The above leads to our approach to TQuel aggregates: to aggregate a given attribute of relation R,

(a) Determine the periods of time during which R remained "fixed" or "constant", that is, no new tuples entered the relation (and, if R is an interval relation, no tuples became invalid).

(b) If there is a by-list with this aggregate, subdivide each constant set of tuples into subsets, each subset corresponding to a value of the by-list attributes.

(c) For each constant set of tuples, select the tuples that satisfy all the qualifications required by the where, when and as-of clauses, if any. Defaults are used if those clauses are not present. Each group of selected tuples is called an *aggregation set*.

(d) Compute the aggregate for each aggregation set. Output the result from each group as a *tuple* valid during its associated period of time, intersected with the interval or event specified by the valid clause.

The basic strategy therefore consists of reducing a TQuel query to a series of Quel-style queries, each applied on a period of time when the relation does not change its contents.

Thus, TQuel queries with aggregates can result in several tuples rather than a single value. Each tuple contains the value of the aggregate, attached to the particular period of time it was valid, showing that the aggregate is really a time-varying function. At each point in time, there is exactly one value of the aggregate. A set of tuples is required to model the *history* of the aggregate.

Quel allows an inner where clause as the way 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-through clauses serve the same purpose.

3.3. New Aggregates

All Quel aggregates have a TQuel counterpart. There are also some aggregates unique to TQuel. The first is quite similar to `avg`, applying both to static relations and temporal relations:

`stdev` The standard deviation of the set of *n* values present in a given attribute, defined as a measure of the homogeneity of the values. This operator is restricted to operate only on numeric attributes.

The remaining new aggregates are strictly temporal.

`first` This instantaneous 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, the one with earlier *to* time is considered to be older. If they have the same *from* and *to* values, one is arbitrarily selected.

`last` This instantaneous aggregate is analogous to `first`. It returns, at each point in time, the newest value of the given attribute, that is, the one associated with the last valid tuple. If two tuples have the same *to* value, the one with later *from* time is considered to

be newer. If they have the same *from* and *to* values, one is arbitrarily selected.

The cumulative versions **firstC** and **lastC** are also available. Note that, while **first**, **last**, and **lastC** yield (potentially) several tuples of output, **firstC** outputs just one tuple.

All these new aggregates operate on the explicit attributes of relations. The next two are useful when analyzing numeric data varying over time.

avgti AVeraGe Time Increment: the average growth or decrease experienced by values of an attribute over time. This aggregate is only applicable to numeric attributes. It returns a value indicating growth per time unit; for example, 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 Section 3.6).

The **avgti** is obtained by comparing the attribute value of each tuple with the attribute value of its chronologically previous tuple, relative to the time elapsed, and smoothing out all the comparisons by taking their arithmetic mean. At least two tuples are needed to compute **avgti** so that the comparison can be made.

varts VARiability of Time Spacing: the degree of inequality of the time spacing within a given set of attribute values. This aggregate returns a non dimensional quantity which has the same value for each attribute. A value of 0 indicates the tuples are perfectly spaced.

The **varts** 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. Like in **avgti**, at least two tuples are needed to perform the comparison.

In addition, two instantaneous and two cumulative aggregates that operate on the implicit valid times are available. They can be employed by the user to specify conditions in the temporal qualification (when clause), the valid times (valid clause), and/or the transaction times (as-of-through clause).

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

latest The newest time period of an interval relation, that is, the last *from-to* interval or *at* event. If two tuples of an interval relation have the same *to* value, the one with later *from* time is considered to be newer.

The cumulative versions **earliestC** and **latestC** are defined as well. They respectively output the earliest and the latest time periods from the beginning of the query interval. Aggregates in the when, valid and as-of clauses are called ***aggregated temporal constructors*** because

they return a time interval as their result. To adhere to the syntax of temporal expressions and predicates, the **earliest**, **latest**, **earliestC**, and **latestC** aggregates take a tuple variable, rather than an attribute, as an argument.

3.4. Some Examples

The first example shows how an aggregate, which gives an interval relation, can occur with an event relation in a query.

```
range of f is Faculty
range of s is Submitted
retrieve (s.Author, s.Journal, NumFac = count(f.Name when f overlap s))
```

Example 4: How many faculty members were there each time a paper was submitted to a journal?

The result is:

Author	*Journal*	*NumFac*	*at*
Merrie	CACM	3	9-78
Merrie	TODS	3	5-79
Jane	CACM	3	11-79
Tom	JACM	3	12-82

The count is computed for every period of time such that **f** overlaps **s**, and then, by default, the valid times of the output are the overlap of the valid times of the count and the **s** tuple variable, producing an event relation.

This query, modified from one given in [Epstein 1979], shows an aggregate in the inner where clause of another aggregate; a case of nested aggregation:

```
range of f is Faculty
retrieve (f.name, tiny = min(f.salary where f.salary != min(f.salary))
    when begin of f precede "1980"
```

Example 5: Who was making the second smallest salary, and how much was it, during each period of time prior to 1980?

The output is

name	*tiny*	*from*	*to*
Tom	23000	9-75	12-76
Tom	23000	12-76	9-77
Merrie	25000	9-77	11-80
Merrie	25000	11-80	12-80

Aggregates can also appear outside the target list:

```
range of f1 is Faculty
range of f2 is Faculty
retrieve (f2.Name, f2.Rank)
    where f1.Name != f2.Name and f1.Rank = f2.Rank
    when earliestC(f1 by f1.Rank) overlap f2
```

Example 6: Who were the professors hired into or promoted to a rank while the first faculty member ever in that rank was still in that category?

Observe that the aggregates in the when, valid and as-of clauses have a tuple variable, rather than an attribute, as argument. First the `earliestC` in each rank is computed,

Rank	*earliestc(f1)*
Assistant	<9-71, 12-76>
Associate	<12-76, 11-80>
Full	<11-80, ∞ >

Only one tuple qualifies, and the output is

Name	*Rank*	*from*	*to*
Tom	Assistant	9-75	12-80

The when clause can be used inside an aggregate:

```
range of f is Faculty
retrieve (amountct = countuUC(f.salary when end of e precede "1981"))
```

Example 7: How many different salary amounts has the department paid its members since its creation until 1981?

Through the use of `countUC`, each salary amount is counted only once for each period of time. The count is nondecreasing since a cumulative operator is specified. The result is

amountct	*from*	*to*
1	9-71	9-75
2	9-75	12-76
3	1-77	9-77
4	9-77	11-80
5	11-80	∞

An instantaneous and a cumulative aggregate can occur simultaneously in a query:

```
range of f is Faculty
range of s is Submitted
retrieve (f.Name,
          topmost = last(f.Rank by f.Name),
          fpaper = firstC(s.Journal by s.Author))
     valid from begin of f to end of f
     where s.Author = f.Name
```

Example 8: For each faculty member, list his/her last rank and first journal they submitted a paper to.

The computation of `last` yields

Name	*last(Rank)*	*from*	*to*
Jane	Full	11-80	∞
Tom	Associate	12-80	∞
Merrie	Associate	12-82	∞

and the computation of `firstC` produces

Author	*firstc(Journal)*	*from*
Jane	CACM	11-79
Merrie	CACM	9-78
Tom	JACM	12-82

Combining the two intermediate relations, we get the final output.

Name	*topmost*	*fpaper*	*from*	*to*
Jane	Full	CACM	11-80	∞
Merrie	Associate	CACM	12-82	∞
Tom	Associate	JACM	12-82	∞

Our last example applies `varts` to the event historical relation *experiment*, which has the following tuples:

experiment(yield):

yield	*at*
1.78	32
1.79	34
1.83	36
1.84	37
1.88	39
1.88	41
1.90	43
1.91	45

```
range of x is experiment
retrieve (VarSpacing = varts(x.yield))
```

Example 9: Given the above set of experimental data, how equally spaced are the observations in time?

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

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

The intermediate calculations, rounded to four decimal places, are displayed in this table:

at	*Time elapsed*	*Coefficient of variation*
32		
34	2	
36	2	0.0000
37	1	0.2828
39	2	0.2474
41	2	0.2222
43	2	0.2033
45	2	0.1884

and the result is the following relation:

VarSpacing	*from*	*to*
0.0000	36	37
0.2828	37	39
0.2474	39	41
0.2222	41	43
0.2033	43	45
0.1884	45	∞

VarSpacing in this case decreases with time. Since *VarSpacing* = 0 means that all tuples are equally time-spaced, the gradual decrease in *VarSpacing* means that the observations, as time passes, are approaching uniformity in their time spacing. The initial 0.0000 says that the first three observations were perfectly time spaced. Because of the number of elements required to compute a standard deviation, *VarSpacing* is not defined before time 36.

3.5. Defaults

The computation of an aggregate is really a query in itself, thus it is natural to use the same defaults in the three inner clauses. For each attribute being aggregated, the defaults must guarantee that all existing tuples in the corresponding relation participate in the aggregation. Hence,

```
where true
when t1 overlap ··· overlap tk
as of "now"
```

These defaults permits the reduction of TQuel aggregates to Quel aggregates to be proven (c.f., [Snodgrass 1986]), thereby allowing TQuel aggregates to be used in exactly the same way as Quel aggregates.

3.6. Syntax Summary

In order to accommodate aggregates, the TQuel syntax [Snodgrass 1986] is slightly augmented. TQuel is a superset of Quel, that is, all legal Quel statements with aggregates are also legal TQuel statements with aggregates. The following are the additions made to the above mentioned TQuel syntax.

```
<expression>         ::= In addition to the TQuel syntax, include:
                     | <aggregate term>
<aggregate term>     ::= <aggregate op> ( <arith expr><by clause><retrieve tail> )
<by clause>          ::= ε | by <attribute list>
<attribute list>     ::= <arith expr> | <attribute list>, <arith expr>
<aggregate op>       ::= count | countC | countU | countUC
                     | sum | sumC | sumU | sumUC
                     | avg | avgC | avgU | avgUC
                     | stdev | stdevC | stdevU | stdevUC
                     | any | anyC
                     | min | minC
                     | max | maxC
                     | first | firstC
                     | last | lastC
                     | <per clause> avgti | per <time unit> avgti
                     | varts
<time unit>          ::= millisecond | second | minute | hour
                     | day | week | month | quarter | year
<interval element>   ::= In addition to the TQuel syntax, include:
                     | <aggt> ( <tuple variable><by clause><retrieve tail> )
<aggt>               ::= earliest | earliestC | latest | latestC
```

Chapter 4

TUPLE CALCULUS SEMANTICS OF TQUEL AGGREGATES

It is convenient to base the semantics of TQuel on the static relational database model, especially because of the available mathematical foundation supporting the latter[Codd 1972]. Thus the semantics of the augmented operations are expressed using traditional tuple calculus notation.

We will 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 from parts of [Snodgrass 1986]. The semantics of the TQuel aggregates is then developed.

4.1. Review of TQuel Semantics

As stated in the overview of TQuel in Chapter 3, 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 (*beginof*, *endof*, *overlap*, *extend*) that take one or two intervals and compute an interval and temporal predicate functions (*precede*, *overlap*) that take two intervals and compute a boolean value. All of them are ultimately defined in terms of the predicate *Before* 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*, $\wedge$, $\vee$, and $\neg$ operations. The valid clause is transformed into the functions Φ_v and Φ_χ, each evaluating to an event, and containing the functions *first* and *last*. The as-of-through 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-through 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 $(t_{i_1}.D_{j_1}, \ldots, t_{i_r}.D_{j_r})$
  valid from υ **to** χ
  where ψ
  when τ
  as of α **through** β

is translated into the tuple calculus statement

$$\Big\{ w^{(r+4)} \mid (\exists t_1) \cdots (\exists t_k)$$
$$(R_1(t_1) \wedge \cdots \wedge R_k(t_k)$$
$$\wedge\ w[1] = t_{i_1}[j_1] \wedge \cdots \wedge\ w[r] = t_{i_r}[j_r]$$
$$\wedge\ w[r+1] = \Phi_\upsilon \wedge w[r+2] = \Phi_\chi \wedge Before(w[r+1], w[r+2])$$
$$\wedge\ w[r+3] = now \wedge w[r+4] = \infty$$
$$\wedge\ \psi'$$
$$\wedge\ \Gamma_\tau$$
$$\wedge\ (\forall l)(1 \le l \le k)(Before(\Phi_\alpha, t_l[stop]) \wedge Before(t_l[start], \Phi_\beta)))$$
$$)\Big\}$$

In this statement, *now* represents the current transaction time. The superscript indicates that the tuple u has r explicit attributes and 4 implicit attributes; this clearly refers to an interval relation. The semantics for an event relation is similar, but with only 3 implicit attributes, since the *to* time is not present.

EXAMPLE. Example 3,

```
range of f1 is Faculty
range of f2 is Faculty
retrieve (f1.Rank)
valid at begin of f1
where f1.Name = "Jane" and f2.Name = "Merrie" and f2.Rank = "Associate"
when f1 overlap begin of f2
```

which results in an event relation, has the following tuple calculus semantics,

$$\Big\{ w^{(1+8)} \mid (\exists f1)(\exists f2)$$

$$(Faculty(f1) \bigwedge Faculty(f2)$$

$$\bigwedge w[1] = f1[Rank]$$

$$\bigwedge w[1+1] = f1[from]$$

$$\bigwedge w[1+2] = now \bigwedge w[1+3] = \infty$$

$$\bigwedge f1[Name] = \texttt{"Jane"} \bigwedge f2[Name] = \texttt{"Merrie"} \bigwedge f2[Rank] = \texttt{"Associate"}$$

$$\bigwedge Before(f1[from], f2[from]) \bigwedge Before(f2[from], f1[to])$$

$$)\Big\} \;\blacksquare$$

4.2. New TQuel Aggregates

Let us specify the semantics of the new aggregates introduced in Section 3.3. Let R be an event relation of degree r (recall that the degree only concerns the explicit attributes) with n tuples, $n>2$. These aggregates all compute a single static tuple of degree r.

DEFINITION.

$$S \triangleq chronorder(R) \Longleftrightarrow (\forall i)(1 \le i \le |S|)\, ((\exists t)\,(R(t) \bigwedge t = S_i)) \bigwedge Before(S_{i-1}[at], S_i[at]) \bigwedge S_{i-1}[at] \neq S_i[at]\,)$$

where $|S|$ is the length of the sequence 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.

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

$$\left(\frac{1}{|S|-1} \sum_{i=1}^{|S|-1} \frac{S_{i+1}[r] - S_i[r]}{S_{i+1}[at] - S_i[at]} \right) \Bigg)$$

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 timestamp granularity, defined in Chapter 3). An optional per clause can be used to specify the time unit desired; this causes multiplication of the result by a fixed conversion factor. For example, if timestamp 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. $varts(R) \triangleq \left(\frac{sd(D(R))}{mean(D(R))}, \ldots, \frac{sd(D(R))}{mean(D(R))} \right)$

where $D(R) \triangleq <d_1, \cdots, d_{|S|-1}>$ such that $S = chronorder(R), |S| > 1$, $(\exists i)(1 \le i \le |S|-1 \wedge d_i = S_{i+1}[at] - S_i[at])$, 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 $D(R)$. The value is the same for all r attributes.

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 *avgti* or *varts* will not attempt a division by zero. Should the user need to specify which of the tuples from R has to be chosen for the chronological order, one of the other aggregates can be used to create a temporary relation T that contains the relevant tuples, and then `avgti` or `varts` may be applied to T.

Let R be an interval relation of degree r, and t be a tuple variable associated with R.
DEFINITION.

$$stdev(R) \triangleq \left(\sqrt{\frac{1}{n} \sum_{t \in R(t)} (t[1])^2 - \frac{1}{n^2} (\sum_{t \in R(t)} t[1])^2}, \ldots, \sqrt{\frac{1}{n} \sum_{t \in R(t)} (t[r])^2 - \frac{1}{n^2} (\sum_{t \in R(t)} t[r])^2} \right)$$

Each component of the result tuple equals the standard deviation of all values in the corresponding component of the tuples of R.

DEFINITION. $firstagg(R) \triangleq (t_{first}[1], \ldots, t_{first}[r])$

where t_{first} represents the tuple such that

$$R(t_{first})$$
$$\wedge\ (\forall t)\ (\ R(t) \Rightarrow Before\ (t_{first}[r+1],\ t[r+1]))$$
$$\wedge\ (\forall t)\ (\ (R(t) \wedge t_{first}[r+1] = t[r+1]) \Rightarrow Before\ (t_{first}[r+2],\ t[r+2])\ )$$

The result tuple equals that tuple whose valid times are the earliest valid times in R. More specifically, the second line of this predicate says that t_{first} began before all other tuples in R, and the third line means that if another tuple from R had the same *from* time as t_{first}, then t_{first} ended before that tuple.

DEFINITION. $lastagg(R) \triangleq (t_{last}[1], ..., t_{last}[r])$

where t_{last} represents the tuple such that

$$R(t_{last})$$
$$\wedge\ (\forall t)\ (\ R(t) \Rightarrow Before\ (t[r+2],\ t_{last}[r+2]))$$
$$\wedge\ (\forall t)\ (\ (R(t) \wedge t_{last}[r+2] = t[r+2]) \Rightarrow Before\ (t[r+1],\ t_{last}[r+1])\ )$$

The result tuple equals that tuple whose valid times are the latest valid times in R. More specifically, the second line of this predicate says that t_{last} ended after all other tuples in R, and the third line means that if another tuple from R had the same *to* time as t_{last}, then t_{last} began after that tuple.

Notice that, like the other aggregate operators, *firstagg* and *lastagg* both yield a single tuple with r explicit attributes. The implicit time attributes will be given later in the complete tuple calculus statements.

Let R be an interval relation, $R[from]$ be the value of the *from* implicit temporal attribute of R, and $R[to]$ be the value of the *to* temporal attribute of R.

DEFINITION. $earliest(R) \triangleq \langle firstagg(R)[from],\ firstagg(R)[to] \rangle$

The result is the interval represented by the valid times of the earliest tuple in the relation.

DEFINITION. $latest(R) \triangleq \langle lastagg(R)[from],\ lastagg(R)[to] \rangle$

The result is the interval represented by the valid times of the latest tuple in the relation.

4.3. The Constant Predicate

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 may look similar to Quel queries with scalar aggregates, in which no inner when or as-of-through clauses exist (recall the default clauses from Chapter 3). 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. The interval 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 in the by list). It is convenient to determine the points at which the value changes. Let us first define the *time-partition* of a set of relations as

$$T(R_1, ..., R_k) \triangleq \left\{ e \mid (\exists t)(\exists i)\,(1 \leq i \leq k \wedge R_i(t) \wedge (e = t[from] \vee e = t[to])) \right\}$$

The time partition brings together all the events e of the relations $R_1, ..., R_k$, that is, all tuple beginnings and endings (if the tuple is from an event relation, only one event is contributed). If two events c and d are neighbors, i.e. no other event occurred between them, the time interval from c to d did not witness any change in the set of relations, or in other words, all the relations remained "constant". Define then the *Constant* predicate as

$$\begin{aligned} Constant(R_1, ..., R_k, c, d) \Longleftrightarrow\; & c \in T(R_1, ..., R_k) \\ & \wedge\, d \in T(R_1, ..., R_k) \\ & \wedge\, c \neq d \\ & \wedge\, Before\,(c, d) \\ & \wedge\, (\forall e)(e \in T(R_1, ..., R_k) \Longrightarrow Before(e, c) \vee Before(d, e)) \end{aligned}$$

In this predicate, the last line means that there is no event in the time between c and d. The constant predicate will allow us to treat each constant time interval $<c, d>$ separately, thus reducing the inner query to a number of queries, each dealing with a constant time interval. In other words, we will be able to follow the same steps as in the static Quel case. For each time interval $<c, d>$ given by the constant predicate a value of the aggregate, valid from c to d, will be computed and will potentially go into the result. This value is guaranteed to be unique by the definition of *Constant*.

EXAMPLE. For the *Faculty* relation, only for the following values of c and d is the $Constant(Faculty, c, d)$ predicate true:

c	d
9-71	9-75
9-75	12-76
12-76	9-77
9-77	11-80
11-80	12-80
12-80	12-82
12-82	*now*

Note that these consecutive intervals are exactly the ones indicated in Figure 1. ∎

4.4. Instantaneous Aggregates

For a multi-relational query with one instantaneous aggregate we will take the approach used in the Quel semantics: tuples from the aggregate operation will be computed first via partitioning functions. Again, let F be any of the aggregate operators defined so far. Consider the TQuel query with one aggregate function,

$$
\begin{array}{l}
\textbf{range of } t_1 \textbf{ is } R_1 \\
\quad \cdots \\
\textbf{range of } t_k \textbf{ is } R_k \\
\textbf{retrieve } (t_{i_1}.D_{j_1}, \ldots, t_{i_r}.D_{j_r}, y = f(t_{l_1}.D_{m_1} \textbf{ by } t_{l_2}.D_{m_2}, \ldots, t_{l_n}.D_{m_n} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where } \psi_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{when } \tau_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{as of } \alpha_1 \textbf{ through } \beta_1 \,)) \\
\quad \textbf{valid from } \upsilon \textbf{ to } \chi \\
\quad \textbf{where } \psi \\
\quad \textbf{when } \tau \\
\quad \textbf{as of } \alpha \textbf{ through } \beta
\end{array}
$$

in which

$$
\begin{array}{l}
1 \le i_1 \le k, \ldots, 1 \le i_r \le k \\
1 \le l_1 \le k, \ldots, 1 \le l_n \le k \\
1 \le j_1 \le deg(R_{i_1}), \ldots, 1 \le j_r \le deg(R_{i_r}) \\
1 \le m_1 \le deg(R_{l_1}), \ldots, 1 \le m_n \le deg(R_{l_n}).
\end{array}
$$

As with Quel, the where predicate should refer only to the tuple variable t_{l_1} or the tuple variables appearing in the by clause. The same restriction holds for the when and as-of clauses

appearing in the aggregate.

Here, the partitioning functions will be based upon the four clauses that modify the aggregate (the by, where, when and as-of clauses). Hence, using the same notation as in Chapter 2,

$$
\begin{aligned}
P\,(a_2, \ldots, a_n, c, d) \triangleq \Big\{ b^{(p)} \mid\; & (\exists t_{l_1}) \cdots (\exists t_{l_n}) \\
& (R_{l_1}(t_{l_1}) \wedge \cdots \wedge R_{l_n}(t_{l_n}) \\
& \wedge\, b = t_{l_1} \\
& \wedge\, t_{l_2}[m_2] = a_2 \wedge \cdots \wedge t_{l_n}[m_n] = a_n \\
& \wedge\, \psi_1{}' \\
& \wedge\, \Gamma_{\tau_1} \\
& \wedge\, (\forall h)(1 \le h \le r)\; Before(t_{i_h}[start], \Phi_{\beta_1}) \wedge Before(\Phi_{\alpha_1}, t_{i_h}[stop]) \\
& \wedge\, Before(t_{l_1}[from], c) \wedge Before(d, t_{l_1}[to]) \\
& \wedge\, (\forall h)(1 \le h \le n)\; Before(t_{l_h}[from], c) \wedge Before(d, t_{l_h}[to]) \\
&) \Big\}
\end{aligned}
$$

where c and d are valid times, with $c < d$ and $p = deg(R_{l_1})$.

This definition assumes that the tuple variables $t_{l_1}, \ldots, t_{l_n}$ are distinct. If they are not, then the duplicate tuple variables should be removed from the first three lines. Line 7 translates the as-of-through 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-through clause. This is similar to the as-of line in the outer query in TQuel. Lines 8 and 9 indicate that the tuple t_{l_r}, associated with the aggregated attribute, and all tuples participating in the by-list must overlap the interval $<c, d>$ (incidently, from the definition of the *Constant* predicate, which will supply the intervals $<c, d>$, 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.

The output relation from a query with an instantaneous aggregate is

$$
\Big\{ w^{((r+1)+4)} \mid (\exists t_1) \cdots (\exists t_k)(\exists c)(\exists d)
$$

$$
(R_1(t_1) \wedge \cdots \wedge R_k(t_k)
$$

$$
\wedge\ Constant(R_{i_1}, \ldots, R_{i_n}, c, d)
$$

$$
\wedge\ w[1] = t_{i_1}[j_1] \wedge \cdots \wedge w[r] = t_{i_r}[j_r]
$$

$$
\wedge\ w[r+1] = f(P(t_{i_2}[m_2], \ldots, t_{i_n}[m_n], c, d))[m_1]
$$

$$
\wedge\ w[r+2] = last(\Phi_v, c) \wedge w[r+3] = first(\Phi_\chi, d) \wedge Before(w[r+2], w[r+3])
$$

$$
\wedge\ w[r+4] = now \wedge w[r+5] = \infty
$$

$$
\wedge\ \psi'
$$

$$
\wedge\ \Gamma_r
$$

$$
\wedge\ (\forall l)(1 \leq l \leq k)\,(Before(\Phi_\alpha, t_l[stop]) \wedge Before(t_l[start], \Phi_\beta))
$$

$$
)\Big\}
$$

A comparison with the tuple calculus expression given in Section 4.1 reveals that lines three and five are new and lines one and six are altered. The *Constant* predicate 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. It ensures that the value of the aggregate, computed in line five, is constant during the interval $<c, d>$(recall that c and d are events delimiting one of the intervals occurring in a relation appearing as a parameter to the *Constant* predicate.) Line six states that the tuple u is valid during the overlap of $<c, d>$and the valid interval specified in the valid-from-to clause. Those portions of the valid interval not accounted for will appear in another tuple, using a different c and d and probably a different value for the aggregate.

EXAMPLE. Let us translate Example 6 operating on an historical database into tuple calculus.

$$P(a_2, c, d) \triangleq \Big\{ b^{(2)} \mid (\exists f)$$

$$(Faculty(f)$$

$$\wedge\ b = f$$

$$\wedge\ f[Rank] = a_2$$

$$\wedge\ Before(f[from], c) \wedge Before(d, f[to])$$

$$) \Big\}$$

P(Assistant, 9-71, 9-75) = {(Jane, Assistant, 25000, 9-71, 12-76)}

P(Assistant, 9-75, 12-76) = {(Jane, Assistant, 25000, 9-71, 12-76),
(Tom, Assistant, 23000, 9-75, 12-80)}

The output relation is

$$\Big\{ w^{(2+2)} \mid (\exists f)(\exists c)(\exists d)$$

$$(Faculty(f)$$

$$\wedge\ Constant(Faculty, c, d)$$

$$\wedge\ w[1] = f[Rank]$$

$$\wedge\ w[2] = count(B(f[Rank], c, d)[Name]$$

$$\wedge\ w[3] = last(f[from], c) \wedge w[4] = first(f[to], d)$$

$$) \Big\}$$

Note it is not necessary to explicitly write $Before(w[3], w[4])$ here, as it was the case when no aggregate was present. ▮

For an aggregate with no by-list, only the where, when and as-of clauses may be present, and the partitioning function P becomes again a subset of R_{i_1}:

$$
\begin{aligned}
P(c,d) = \Big\{ t^{(p)} \mid \; & (\exists t_1) \cdots (\exists t_k) \\
& (R_1(t_1) \wedge \cdots \wedge R_k(t_k) \\
& \wedge t = t_{l_1} \\
& \wedge \psi_1{}' \\
& \wedge \Gamma_{\tau_1} \\
& \wedge (\forall h)(1 \leq h \leq r)\, (\mathit{Before}(t_{i_h}[\mathit{start}], \Phi_{\beta_1}) \wedge \mathit{Before}(\Phi_{\alpha_1}, t_{i_h}[\mathit{stop}])) \\
& \wedge \mathit{Before}(t_{l_1}[\mathit{from}], c) \wedge \mathit{Before}(d, t_{l_1}[\mathit{to}]) \\
&) \Big\}
\end{aligned}
$$

The tuple calculus statement for the query remains the same as above, except that $P(c, d)$ is used in place of $P(t_{l_2}[m_2], ..., t_{l_n}[m_n], c, d)$ and only R_{l_1}, c, and d are needed as arguments to the *Constant* predicate.

Once again, in the case of a multi-aggregate query, say $f_1, ..., f_h$, a separate partitioning function P of either the by-list or the non-by list form has to be defined for each aggregate. The *Constant* predicate should mention the relations associated with all the tuple variables appearing in any aggregate in the query.

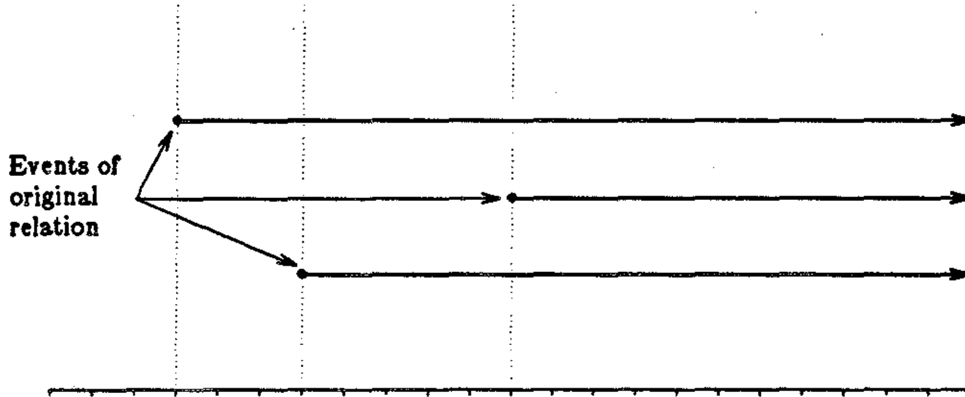
4.5. Cumulative Aggregates

In TQuel, cumulative aggregates can be defined for both event and interval relations. A cumulative aggregate operator applied on an event or an interval relation computes a function f on all tuples that have been valid prior to *now*.

The instantaneous aggregates have their cumulative peers. They perform the same operations, but in an additive fashion as far as tuple validity is concerned, that is, assuming at all times that the tuples created so far are still valid. It is possible to compute them by taking advantage of the already defined instantaneous aggregates. To do this, any event relation R may

be converted into an interval relation by defining all tuples in the interval relation to begin at R's event times and to end in the infinite. Then aggregates can be computed as in the instantaneous case over this interval relation *induced by R*, as shown below:

Figure 4: Converting an Event Relation into an Interval Relation



If R is an event relation of degree r, then its induced stretch is given by the interval relation,

DEFINITION.

$$I^{e}(R) = \Big\{ t^{(r+4)} \mid (\exists t') \\ (R(t') \\ \wedge\ t[1] = t'[1] \wedge \cdots \wedge t[r] = t'[r] \\ \wedge\ t[r+1] = t'[r+1] \wedge t[r+2] = \infty \\ \wedge\ t[r+3] = t'[r+2] \wedge t[r+4] = t'[r+3]\ ) \Big\}$$

A tuple is added to $I^{e}(R)$ at the time a new tuple enters R, and no tuple is added to or deleted from $I^{e}(R)$ at other times. Thus a cumulative aggregate will change its value at the time a new tuple is added to the relation, and will remain constant at all other times.

Now consider an interval relation. Cumulative aggregates depend on the beginning points of tuples. That is, they change their value whenever a new tuple begins, and their value is unaffected when a tuple ends. This suggests how to define an induced stretch for interval rela-

tions. Tuples in it will have valid times beginning at R's beginning times and ending in the infinite. If R is an interval relation of degree r, then its induced stretch is given by the interval relation

DEFINITION.

$$I'(R) \triangleq \Big\{ t^{(r+4)} \mid (\exists t') \\ (R(t') \\ \wedge\ t[1] = t'[1] \wedge \cdots \wedge t[r] = t'[r] \\ \wedge\ t[r+1] = t'[r+1] \wedge t[r+2] = \infty \\ \wedge\ t[r+3] = t'[r+3] \wedge t[r+4] = t'[r+4]\ ) \Big\}$$

EXAMPLE. The induced stretch of *Faculty* is

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

∎

With these definitions, the same time partition and constant predicate as for event relations can be employed.

Let f represent the cumulative version of any of the aggregate operators defined thus far, namely, `anyC`, `countC`, `sumC`, `avgC`, `stdevC`, `maxC`, `minC`, `firstC`, or `lastC`. The general query will be exactly the same as for interval relations. To obtain the aggregation sets, the appropriate I should be computed from R_{l_1} and then used instead of R_{l_1} in each of them. An example will be given shortly.

The output from a cumulative aggregate is also an interval relation, because computed aggregate values are valid during intervals of time.

It is interesting to note that **`first`** may change over time, because the set of tuples comprising an interval relation may change over time. **`firstC`**, on the other hand, stays the same. Another, perhaps obvious, fact is that the same tuple results from applying either **`last`** or **`lastC`** on an interval relation.

The semantics of **`avgti`** and **`varts`** is the same as that of the other cumulative aggregates. Partitioning functions and the *Constant* predicate are used with or without the $R_{l_2}, \ldots, R_{l_n}$ list depending on whether or not the query contains a by-list.

EXAMPLE. This is the tuple calculus version of Example 9 from Chapter 3.

$$P(c, d) \triangleq \Big\{ t^{(1)} \mid (\exists x)\,(x \in I^{e}(experiment) \\ \wedge\; t = x \\ \wedge\; Before(x[from], c) \wedge Before(d, x[to]) \\ \Big\}$$

$$\Big\{ w^{(1+e)} \mid (\exists x)(\exists c)(\exists d)(\exists t) \\ (experiment(x) \wedge t \in I^{e}(experiment) \\ \wedge\; Constant(I^{e}(experiment), c, d) \\ \wedge\; w[1] = varts(P(c, d))[yield] \\ \wedge\; w[2] = last(t[from], c) \wedge w[3] = first(d, t[to]) \\ ) \Big\}$$

Note the tuple variable x appears only within the aggregate. Thus $\exists x$ and $experiment(x)$ can be omitted from the tuple calculus statement in this case. The last line provides the default valid clause. ∎

4.6. Mixing Different Aggregates in a Query

A TQuel query may call for several aggregates, some of them instantaneous and some others cumulative. Of course, each of the aggregates is computed from its own partitioning functions. When each of the partitioning functions refers to a different set of relations, the *Constant* predicate takes as arguments the relations in all partitioning functions. A simpler procedure, however, is to take all the relations in the query.

Valid times for each output tuple are computed by following the same approach as before: each output tuple is valid during an interval when tuples from all the non-aggregate attributes are in the $<\Phi_\upsilon,\Phi_\chi>$ interval, and this interval overlaps the valid times of the calculated aggregates.

4.7. Aggregates in the Outer Where Clause

TQuel aggregates, or arithmetic expressions containing TQuel aggregates, may be part of the main where or when clause.

EXAMPLE. Example 6 illustrates this point. Let I be the induced stretch of *Faculty*.

$$P(a_2,\ c,\ d) \triangleq \Big\{ t^{(4)} \mid (\exists f)\,(f \in I'(Faculty) \\ \wedge\ t = f \\ \wedge\ f[Rank] = a_2 \\ \wedge\ Before(f[from],\ c) \wedge Before(d,\ f[to]) \Big\}$$

Actually, since $f[to] = \infty$ for all $f \in I'(Faculty)$, the last *Before* is not necessary.

$$P(\text{Assistant, 9-71, 9-75}) = \{(\text{Jane, Assistant, 25000, 9-71, 12-76})\}$$

The relation resulting from the query is

$$
\Big\{ w^{(2+2)} \mid (\exists f1)(\exists f2)(\exists c)(\exists d)
$$

$$
(Faculty(f) \wedge Faculty(f2)
$$

$$
\wedge\ Constant(I'(Faculty), c, d)
$$

$$
\wedge\ w[1] = f2[Name] \wedge w[2] = f2[Rank]
$$

$$
\wedge\ w[3] = last(f1[from], last(f2[from], c)) \wedge w[4] = first(f1[to], first(f2[to], d))
$$

$$
\wedge\ f[Name] \neq f2[Name] \wedge f1[Rank] = f2[Rank]
$$

$$
\wedge\ Before(earliest(P(f1[Rank], c, d))[from], f2[to])
$$

$$
\wedge\ Before(f2[from], earliest(P(f1[Rank], c, d))[to])
$$

$$
)\Big\}
$$

The fifth line originates from the default valid clause, which in this case is

```
valid from begin of (f1 overlap f2) to end of (f1 overlap f2)
```

Note that the instantaneous *earliest* is used. The fact that the cumulative version of the aggregate was specified in the TQuel query is reflected in the use of $I'(Faculty)$ in the definition of $P(c, d)$ and its presence in the constant predicate. ∎

Through the partitioning functions, the values of the aggregated attribute are first computed, then used in place of the aggregate in the predicate of the query. Since the variables in by-lists are "global", its by clause is linked to the rest of the query, as in Quel.

4.8. Nested Aggregation

In nested aggregation, the local where clause of an aggregate f_1 invokes another aggregate f_2. If f_2 has a by-list, links are established between the tuple variables in the by-list of f_2 and the tuple variables in the f_1 query.

EXAMPLE. Example 5 contains a nested aggregate. Let us show the partitioning functions P_1 and P_2 for the outer and the inner aggregates respectively:

$$P_2(c, d) = \Big\{ t^{(4)} \mid (\exists f)\,(Faculty(f)$$

$$\wedge\; t = f$$

$$\wedge\; Before(f[from], c) \wedge Before(d, f[to])$$

$$)\Big\}$$

$$P_1(c, d) = \Big\{ t^{(4)} \mid (\exists f)\,(Faculty(f)$$

$$\wedge\; t[1] = f[1] \wedge \cdots \wedge t[4] = f[4]$$

$$\wedge\; f[salary] \neq min(P_2(c, d))[salary]$$

$$\wedge\; Before(f[from], c) \wedge Before(d, f[to])$$

$$)\Big\}$$

The tuple calculus statement for the retrieve statement will contain $P_1(c, d)$ but not $P_2(c, d)$; that is, only one level of nesting occurs at any one time in a tuple calculus statement. ■

4.9. Unique Aggregation

Unique aggregation is also possible in TQuel. There are four instantaneous unique aggregates: `countU`, `sumU`, `avgU`, and `stdevU`, and four cumulative versions of the same: `countUC`, `sumUC`, `avgUC`, and `stdevUC`. It is not necessary to define unique versions for `any`, `max`, `min`, `first`, `last`, `avgti` and `varts`, or their cumulative counterparts, because the same results can be obtained with the non-unique aggregates.

Let p and q be as usual. When the inner query has a by-list, the modified partitioning function is defined in terms of the ordinary P as

$$U(a_2, \ldots, a_n, c, d) = \Big\{ w^{(1)} \mid (\exists b)(b \in P(a_2, \ldots, a_n, c, d) \wedge w[1] = b[m_1]) \Big\}$$

With no by-list, the modified partitioning function $U(c, d)$ is similarly defined from $P(c, d)$.

The simple substitution of U for P in the final tuple calculus statement, together with the use of the non-unique versions of the aggregates, yields the tuple calculus semantics of unique aggregates.

EXAMPLE. To obtain the tuple calculus expression for the aggregation set in Example 7, the induced stretch I is obtained from *Faculty*, P is defined from I, and then

$$U(c, d) = \left\{ w^{(4+2)} \mid (\exists b)(b \in B(c, d) \wedge w[1] = b[salary]) \right\} \quad \blacksquare$$

4.10. Aggregates in the Other Outer Clauses

Four aggregates may be used in the when, as-of, and valid clauses: **earliest**, **latest**, **earliestc**, and **latestc**. Just like in the case of aggregates in the where clause, an aggregate that is used in the when clause can be modified with inner by, where, when and as-of clauses.

With these restrictions, the semantics of the aggregated temporal constructors is the same as that of the other aggregates. For the linking of tuple variables, the same notes as in the outer and inner where clause apply. Being based on *first* and *last* (c.f., Section 4.1), there is no need to define unique versions of the aggregated temporal constructors.

As in the case of **first**, for the **earliest** aggregate, note that, as the composition of an interval relation is time-dependent, **earliest** of an interval relation may also change over time. Moreover, as in the case of **last** and **lastC**, the **latest** and **latestC** aggregates always produce the same result from an interval relation.

Chapter 5

CONCLUSION

We first defined the tuple calculus semantics of Quel aggregates. This definition, coupled with that of the core language [Snodgrass 1986], provides a complete formal semantics of Quel. To the time-oriented aggregates corresponding to the ones already available in Quel, we defined new operators that permit summarization over the time dimension. We then constructed a formal semantics for aggregates in the retrieve statement of the TQuel language.

We started by introducing the *Constant* predicate and the partitioning function. Within intervals computed by *Constant*, a relation remains static, and aggregates can be computed in the way shown in Chapter 2. This enabled us to formally define the semantics for instantaneous aggregates. Later, the introduction of the induced stretch, which transforms any relation into an interval relation ending in the infinite, permitted us to conveniently specify the semantics of cumulative aggregates as a special case of instantaneous aggregates. The issues of freely mixing different aggregates, as well as the semantics of aggregates in the outer where and in the inner where clauses (nested aggregation), were discussed and resolved. When appropriate, unique versions of the aggregates are also provided. For the when, as-of, and valid clauses, the aggregated temporal constructors **earliest** and **latest**, with the corresponding cumulative versions, are available.

It is easy to extend the semantics to specify the TQuel modification statements (**append**, **delete**, and **replace**) to include aggregates, using the strategy discussed in Chapter 4. It is also straightforward to extend the aggregates to include arbitrary expressions, using the technique discussed in Section 2.7.

The result is a complete formal semantics for TQuel and its static subset Quel. A complete formal semantics for no other relational query language, temporal or otherwise, has been defined. The specifications of several other languages come close: the semantics of SQL [Ceri & Gottlob

1985] includes almost all of the SELECT statement, including aggregates, but no modification statements; HTQuel [Gadia & Vaishnav 1985] and Tansel's algebraic language [Clifford & Tansel 1985] do not include modification statements nor aggregate operators.

The next step is to develop an operational semantics in terms of a temporal relational algebra (c.f., [McKenzie 1986]). Here the challenge is the language core, rather than the aggregate functions, which can be added quite easily. Implementation techniques, such as those developed for Quel [Epstein 1979], need to be developed for temporal aggregates.

Throughout this work we have seen that TQuel aggregates can be specified in a natural way, consistent with the core of the query language, and with minimal additions to both the language definition and to its semantics. The semantics of the TQuel aggregates let the DBMS handle the implicit time attributes, consistent with the rest of TQuel. The presence of the time dimension, while adding some complexity to the specification and handling of aggregates, provides the user a rich set of functions capable of extracting information from the database at each point of time or across time.

Acknowledgements

We are grateful to Dr. Peter Bloomfield for his remarks on the requirements of experimental data in statistical time series that lead to the creation of the `varts` operator and to Ilsoo Ahn, David Beard and Ed McKenzie for helpful comments on this paper.

REFERENCES

[Anderson 1982] Anderson, T. L. *Modeling Time at the Conceptual Level.* in *Improving Database Usability and Responsiveness,* Ed. P. Scheuermann. Jerusalem, Israel: Academic Press, 1982, pp. 273-297.

[Ariav 1985] Ariav, G. *A Temporally Oriented Data Model.* Technical Report. New York University. Mar. 1985.

[Ben-Zvi 1982] Ben-Zvi, J. *The Time Relational Model.* PhD. Diss. UCLA, 1982.

[Bradley 1978] Bradley, J. *Operations Data Bases,* in *Proceedings of the Fourth International Conference on Very Large Data Bases,* West Berlin, Germany: Sep. 1978, pp. 164-176.

[Bubenko 1977] Bubenko, J. A., Jr. *The Temporal Dimension in Information Modeling,* in Architecture and Models in Data Base Management Systems. North-Holland Pub. Co., 1977.

[Ceri & Gottlob 1985] Ceri, S. and G. Gottlob. *Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. IEEE Transactions on Software Engineering,* SE-11, No. 4, Apr. 1985, pp. 324-345.

[Clifford & Warren 1983] Clifford, J. and D. S. Warren. *Formal Semantics for Time in Databases. ACM Transactions on Database Systems,* 8, No. 2, June 1983, pp. 214-254.

[Clifford & Tansel 1985] Clifford, J. Tansel, A.U. *On An Algebra For Historical Relation Databases: Two Views. Proceedings of ACM-SIGMOD 1985 International Conference on Management of Data,* , May 1985, pp. 247-265.

[Codd 1972] Codd, E. F. *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 .

[Codd 1979] Codd, E.F. *Extending the Database Relational Model to Capture More Meaning. ACM Transactions on Database Systems,* 4, No. 4, Dec. 1979, pp. 397-434.

[Date 1983] Date, C. J. *An Introduction to Database Systems.* Vol. II of Addison-Wesley Systems Programming Series. Reading, MA: Addison-Wesley Pub. Co., Inc., 1983.

[Epstein 1979] Epstein, R. *Techniques for Processing of Aggregates in Relational Database Systems.* UCB/ERL M7918. Computer Science Department, University of California, Berkeley. Feb. 1979.

[Gadia & Vaishnav 1985] Gadia, S.K. and J.H. Vaishnav. *A Query Language For A Homogeneous Temporal Database,* in *Proceedings of the Conference on Principles of Database Systems,* Apr. 1985.

[Held et al. 1975] Held, G.D., M. Stonebraker and E. Wong. *INGRES--A relational data base management system. Proceedings of the 1975 National Computer Conference,* 44 (1975)

pp. 409-416.

[SQL/DS 1981] IBM *SQL/Data-System, Concepts and Facilities.* Technical Report GH24-5013-0. IBM. Jan. 1981.

[Jones & Mason 1980] Jones, S. and P. J. Mason. *Handling the Time Dimension in a Data Base.* in *Proceedings of the International Conference on Data Bases,* Ed. S. M. Deen and P. Hammersley. British Computer Society. University of Aberdeen: Heyden, July 1980, pp. 65-83.

[Klug 1982] Klug, A. *Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. Journal of the Association of Computing Machinery,* 29, No. 3, July 1982, pp. 699-717.

[LBL 1981] *Proceedings of the First International Workshop on Statistical Database Management.* Ed. H.K. Wong, 1981.

[LBL 1983] *Proceedings of the Second International Workshop on Statistical Database Management.* Ed. J. McCarthy, 1983.

[Lum et al. 1984] Lum, V., P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner and J. Woodfill. *Designing DBMS Support for the Temporal Dimension.* in *Proceedings of the Sigmod '84 Conference,* June 1984, pp. 115-130.

[McKenzie 1986] McKenzie, L.E. and R. Snodgrass. *An Incremental Temporal Relational Algebra.* 1986. (In preparation.)

[Ozsoyoglu, et al. 1986] Ozsoyoglu, G., Z.M. Ozsoyoglu and V. Matos. *Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions.* Technical Report. Department of Computer Engineering and Science, Case Western Reserve University. 1986.

[Sernadas 1980] Sernadas, A. *Temporal Aspects of Logical Procedure Definition. Information Systems,* 5 (1980) pp. 167-187.

[Snodgrass 1982] Snodgrass, R. *Monitoring Distributed Systems: A Relational Approach.* PhD. Diss. Computer Science Department, Carnegie-Mellon University, Dec. 1982.

[Snodgrass 1986] Snodgrass, R. *A Temporal Query Language. Transactions on Database Systems (to appear)* (1986).

[Snodgrass & Ahn 1986] Snodgrass, R. and I. Ahn. *Temporal Databases. Computer (to appear),* (1986).

[Stonebraker et al. 1976] Stonebraker, M., E. Wong, P. Kreps and G. Held. *The Design and Implementation of INGRES. ACM Transactions on Database Systems,* 1, No. 3, Sep. 1976, pp. 189-222.

[Ullman 1982] Ullman, J.D. *Principles of Database Systems, Second Edition.* Potomac, Maryland: Computer Science Press, 1982.