An Overview of TQuel

Richard Snodgrass

Chapter 6

An Overview of TQuel

Richard Snodgrass

6.1 Introduction

In this chapter we discuss the temporal query language TQuel. TQuel is a minimal extension to Quel [Held et al. 1975], the query language for Ingres [Stonebraker et al. 1976]. TQuel supports valid time and transaction time. Unlike many other temporal query languages, it supports aggregates, valid-time indeterminacy (where it is not known exactly when an event occurred), database modification, and schema evolution. We discuss all of these aspects, first informally through examples, then formally by presenting their tuple calculus semantics.

It is impossible to comprehensively cover all the technical aspects of this language in one chapter. Hence, we emphasize intuition, and escew details, replacing them with pointers to the archival literature. Some important features, such as the semantics of the TQuel update statements, the semantics of schema evolution in the algebra, and performance modeling of temporal queries, will get especially short shrift. All the details are available in the referenced papers.

We start with the language itself, building incrementally from the core constructs to the more advanced features, in almost three dozen example TQuel statements. We follow the same approach in presenting the formal semantics. We present tuple calculus equivalents both for generic TQuel statements and for several examples.

TQuel is based on the predicate calculus. To execute a query, a procedural equivalent is required. We define a temporal algebra, again incrementally, and give several important properties, such as closure, completeness, and reducibility to the snapshot algebra. We also show how each TQuel statement can be mapped into the algebra. Finally, we discuss two important topics in implementing the temporal algebra: query optimization, specifically the applicability of existing optimization strategies, and the physical structure of pages storing temporal tuples. The chapter ends with a BNF syntax of TQuel that incorporates all of these features.

6.2 The Language

6.2.1 Data Model

TQuel supports both valid time and transaction time [Snodgrass & Ahn 1985, Snodgrass & Ahn 1986]. As with most conventional query languages, it also supports user-defined time. Tuples are optionally timestamped with either a single valid timestamp (if the relation models events) or a pair of valid timestamps (if the relation models intervals), and optionally a pair of transaction timestamps, denoting when the tuple was logically inserted into the relation and when it was logically deleted. A transaction timestamp of "until changed" indicates that the tuple has not yet been deleted.

Throughout this chapter we utilize a Stock Broker database. This database has been simplified and configured to illustrate many of the features of TQuel, and hence is not entirely realistic. The database contains two relations, `Stocks`, specifying the price of each stock, and `Own`, specifying the stocks owned at any point in time by each client of the stock broker.

EXAMPLE 1 *Define the two relations in the Stock Broker database.*

```
create interval Stocks(Stock is char, Price is monetary)

create persistent interval Own(Client is char, Stock is char,
          Shares is I4)
```
□

`Stocks` is a valid-time interval relation (indicated by the keyword *`interval`*), with two valid-time timestamps. The `Price` is *stepwise constant* [Segev & Shoshani 1987] across the interval. The key is the `Stock` attribute, a *time-invariant* attribute [Navathe & Ahmed 1989]. The `Stocks` relation is updated in real-time by a direct feed from the New York Stock Exchange. Whenever the price of a stock changes, the database is updated immediately. Using the extended taxonomy we previously developed [Jensen & Snodgrass 1992A, Jensen & Snodgrass 1992B], this relation may be classified as a *degenerate bitemporal relation*, in which the valid and transaction times are exactly correlated. In such relations, storing two pairs of timestamps is redundant, so instead we store only one pair of timestamps, effectively representing both valid and transaction time. This aspect is not included in the *`create`* statement as such; the user is responsible for considering this correspondence when specifying queries.

`Stocks`:

Stock	*Price*	*From*	*To*
IBM	8	6-3-91 11:23AM	10-1-91 9:30AM
IBM	10	10-1-91 9:30AM	10-2-91 2:15PM
IBM	12	10-2-91 2:15PM	10-7-91 10:07AM
IBM	16	10-7-91 10:07AM	10-15-91 4:35PM
IBM	19	10-15-91 4:35PM	10-30-91 4:57PM
IBM	16	10-30-91 4:57PM	11-2-91 12:53PM
IBM	25	11-2-91 12:53PM	11-5-91 2:02PM
IBM	27	11-5-91 2:02PM	12-3-91 8:44AM
IBM	23	12-3-91 8:44AM	12-29-91 9:01AM
IBM	21	12-29-91 9:01AM	1-3-92 4:28PM
IBM	23	1-3-92 4:28PM	forever

`Own` is a bitemporal interval relation. The valid timestamp states when the buy and sell actions became effective, and the transaction timestamp (indicated by the keyword ***persistent***) states when the information was recorded in the database. `Shares`, a stepwise constant attribute, is the total number of that stock owned by the client at any point in time. The key is the composite of the time-invariant attributes `Client` and `Stock`.

The chain of events of a buy or sell transaction is somewhat more complicated than updating the price of a stock. The client submits a buy or sell request to the broker, who sends a message to the representative on the floor of the stock exchange. When the transaction actually occurs, it is assigned a valid time. The amount of the transaction is the price of the stock valid at that time, multiplied by the number of shares changing hands. Due to processing that occurs within the stock exchange computer, as well as batch processing of transactions that occurs in the stock broker computer, there is a delay of up to 24 hours before the transaction is actually recorded in the stock broker database. Using the extended taxonomy previously mentioned, `Own` is a *bounded retroactive bitemporal relation*, with the recording bounded to no more than 24 hours after the buy or sell occurred.

`Own`:

Client	*Stock*	*Shares*	*From*	*To*	*Start*	*Stop*
Melanie	IBM	20	9-15-91 3:12PM	forever	9-16-91 2:01AM	12-31-91 3:12AM
Melanie	IBM	20	9-15-91 3:12PM	12-30-91 10:31AM	12-31-91 3:12AM	until changed
Melanie	IBM	30	12-30-91 10:31AM	forever	12-31-91 3:12AM	until changed

Melanie bought her first shares of IBM stock on September 15, 1991 (recorded some eleven hours later in the stock broker database), then purchased an additional ten shares on December 30 of that year, recorded almost 17 hours later.

6.2.2 Quel Retrieval Statements

EXAMPLE 2 *What stocks does Melanie currently own?*

```
range of O is Owner
retrieve (O.Client, O.Stock, O.Shares)
where O.Client = "Melanie"
```

The *target list* specifies the attributes of the retrieved tuples, and the where clause restricts the underlying tuples that participate in the query. This TQuel query yields the same result as its Quel counterpart, in this case that Melanie owns 30 shares of IBM stock. □

Multiple tuple variables can appear in a query.

EXAMPLE 3 *What is the current worth of Melanie's stocks?*

```
range of S is Stocks
retrieve (O.Stock, Value = O.Shares * S.Price)
where O.Client = "Melanie" and O.Stock = S.Stock
```
□

The defaults for the TQuel constructs not present in Quel were chosen to ensure reducibility to Quel (Section 6.3.8), so that a user's intuitive understanding of the semantics of Quel would transfer directly to TQuel.

6.2.3 Transaction Timeslice

The *as of* clause rolls back a transaction-time relation (consisting of a sequence of snapshot relation states) or a bitemporal relation (consisting of a sequence of valid-time relation states) to a particular state as it was stored in the past, as of a specified transaction time. It can be considered to be a transaction time analogue of the where clause, restricting the underlying tuples that participate in the query.

EXAMPLE 4 *What is the current worth of stocks presently owned by Melanie?*

```
retrieve (O.Stock, Value = O.Shares * S.Price)
where O.Client = "Melanie" and O.Stock = S.Stock
as of present
```

Actually, this is the default as of clause, retrieving the best-known information. □

TQuel allows temporal constants (events, intervals, and spans) to be used. These constants can be specified using a variety of calendars and natural languages [Soo & Snodgrass 1992A, Soo & Snodgrass 1992B, Soo et al. 1992].

EXAMPLE 5 *Specify the Gregorian calendar as used in the United States, with English as the language.*

```
set calendric system UnitedStates
```
□

The effect of the *set* statement, like that of the *range* statement, extends to the next such statement that overrides it.

EXAMPLE 6 *What stocks were shown on Melanie's summary of all stocks currently owned, printed at noon on December 30, 1991?*

```
retrieve (O.all)
where O.Client = "Melanie"
as of |12PM Dec 30, 1991|
```

This query uses a temporal event constant, delimited with vertical bars, "|···|". `O.all` is syntactic sugar denoting all of `O`'s attributes. This query yields the following result.

Client	*Stock*	*Shares*	*From*	*To*
Melanie	IBM	20	9-15-91 3:12PM	forever

Note that this summary does not show all stocks purchased on that day, as the purchase at 10:31 that morning was not recorded until early the next day. □

6.2.4 Valid-time Selection

The *when* clause is the valid-time analogue of the where clause: it specifies a predicate on the event or interval timestamps of the underlying tuples that must be satisfied for those tuples to participate in the remainder of the processing of the query.

EXAMPLE 7 *What stocks were owned by Melanie at noon on December 30, 1991 (as best known right now)?*

```
retrieve (O.all)
where O.Client = "Melanie"
when O overlap |12PM Dec 30, 1991|
as of present
```

Client	*Stock*	*Shares*	*From*	*To*
Melanie	IBM	30	12-30-91 10:31AM	forever

A careful examination of the prose statement of this and the previous query illustrates the fundamental difference between valid time and transaction time. The `as of` clause selects a particular transaction time, and thus *rolls back* the relation to its state stored at the specified time. Corrections stored after that time will not be incorporated into the retrieved result. The particular `when` statement given here selects the facts *valid in reality* at the specified time. All corrections stored up to the time the query was issued are incorporated into the result. In this case, we now know that Melanie had purchased an additional 10 shares of IBM stock about an hour and a half before noon on December 30. □

EXAMPLE 8 *What stocks were owned by Melanie at noon on December 30, 1991, as best known at that time?*

```
retrieve (O.all)
where O.Client = "Melanie"
when O overlap |12PM Dec 30, 1991|
as of |12PM Dec 30, 1991|
```

The result of this query, executed any time after noon on December 30, 1991, will be identical to the result of the first query specified, "*What stocks does Melanie currently own?*", executed exactly on noon of that date, indicating **20** shares of IBM stock. □

The predicate in the when clause can be defined over the events and intervals associated with several tuple variables.

EXAMPLE 9 *What was the worth of Melanie's stocks over time?*

```
retrieve (O.Stock, Value = O.Shares * S.Price)
where O.Client = "Melanie" and O.Stock = S.Stock
when O overlap S
```

Because `as of present` is assumed, this query returns the best known information. □

EXAMPLE 10 *What is the current worth of Melanie's stocks?*

```
retrieve (O.Stock, Value = O.Shares * S.Price)
where O.Client = "Melanie" and O.Stock = S.Stock
when O overlap S overlap present
```

This **when** clause selects only the currently valid tuples. This query is identical in meaning to the Quel query presented in Example 3 in Section 6.2.2. □

EXAMPLE 11 *List all the stocks that doubled in price over a period of a month.*

```
range of S2 is Stocks

retrieve (S2.Stock, S2.Price)
where S.Stock = S2.Stock and S2.Price >= 2 * S.Price
when (end of S + %1 month%) overlap S2
```

There is a lot going on in this query, so let's take it step by step. First, `%1 month%` is a *span*, an unanchored length of time [Soo & Snodgrass 1992B]. Spans can be created by taking the difference of two events; spans can also be added to an event to obtain a new event. The tuple variable `S` represents the stock at its original price; `S2` represents the stock after it had doubled in price, which must be within a month of `S`. The query evaluates to two tuples.

Stock	*Price*	*From*	*To*
IBM	25	11-2-91 12:53PM	11-5-91 2:02PM
IBM	27	11-5-91 2:02PM	11-7-91 10:07AM

After November 7, 1991, the price had no longer doubled over the past month (it jumped to $12 per share on October 7). □

While the previous query does illustrate various aspects of the **when** clause, it is nevertheless not very concise. We'll see a much simpler version shortly.

EXAMPLE 12 *List all the stocks that doubled in price over a period of a month, when in Melanie's hands.*

```
retrieve (S2.Stock, S2.Price)
where S.Stock = S2.Stock and S2.Price >= 2 * S.Price
    and S.Stock = O.Stock and O.Client = "Melanie"
when (end of S + %1 month%) overlap S2
    and O overlap end of S and O overlap begin of S2
```

The `O` tuple variable ensures that the stock was owned by Melanie while it was doubling in price (we use a single tuple variable to ensure that Melanie didn't sell and reacquire the stock during this exciting period). □

6.2.5 Valid-time Projection

The **valid** clause serves the same purpose as the target list: specifying the value of an attribute in the derived relation. In this case, the valid time of the derived tuple is being specified.

EXAMPLE 13 *When was IBM stock purchased?*

```
retrieve (S.Price)
valid at begin of S
where S.Stock = "IBM"
```

This query extracts relevant events from an interval relation. □

EXAMPLE 14 *What is the current worth of Melanie's stocks?*

```
retrieve (O.Stock, Value = O.Shares * S.Price)
valid during O overlap S
where O.Client = "Melanie" and O.Stock = S.Stock
when (O overlap S) overlap present
as of present
```

Stock	*Value*	*From*	*To*
IBM	690	1-3-92 4:28PM	forever

This query employs all the defaults implicit in the query of Example 3 in Section 6.2.2. We'll give the formal semantics for this query in Section 6.3.4 and its algebraic equivalent in Section 6.4.8. □

6.2.6 Aggregates

As TQuel is a superset of Quel, all Quel aggregates are still available [Snodgrass, et al. 1992].

EXAMPLE 15 *How many shares of stock does Melanie own?*

```
retrieve (sum(O.Shares where O.Client = "Melanie"))
```

An algebraic version of this query appears in Section 6.4.4. □

EXAMPLE 16 *What is Melanie's current worth on Wall Street?*

```
retrieve (sum(O.Shares * S.Price
            where O.Client = "Melanie" and O.Stock = S.Stock))
```
□

These queries applied to bitemporal relations yield the same result as their conventional analogues, that is, a single value. With just a little more work, we can extract their time-varying behavior.

EXAMPLE 17 *How has Melanie's current worth flucuated over time?*

```
retrieve (sum(O.Shares * S.Price
            where O.Client = "Melanie" and O.Stock = S.Stock))
when true
```
□

New, temporally-oriented aggregates are also available in TQuel. One of the most useful computes the rate of increase (or decrease) over a specified unit of time.

EXAMPLE 18 *What is Melanie's quarterly return on investment?*

```
retrieve (S.Stock, Return=rate(O.Shares * S.Price by S.Stock
                   where O.Client = "Melanie" and O.Stock = S.Stock
                       per %quarter%))
```

Such aggregates may appear wherever a floating point expression is allowed. The *by* clause is from Quel; it partitions the `Stocks` relation into sets with identical values for the `Stock` attribute, then applies the aggregate to each. The *per* clause is specific to the `rate` aggregate. This query is somewhat simplistic, in that it assumes no new investments were made during the quarter. □

The `rate` aggregate allows us to succinctly specify that fairly torturous query in Example 12 in Section 6.2.4.

EXAMPLE 19 *What stocks have doubled in price over the last month?*

```
retrieve (S.Stock, S.Price)
where rate(S.Price by S.Stock for each %month%) >= 2
```

The *for each* clause specifies a *moving window* aggregate [Navathe & Ahmed 1989]. Conceptually, the aggregate is evaluated for each point in time, taking into consideration the values over the month-long interval terminating at that point in time. We'll give the formal semantics for this query in Section 6.3.5 and its algebraic equivalent in Section 6.4.8. □

The `rising` aggregate returns an interval when the argument was rising in value. This aggregate may be used wherever an interval expression is expected.

EXAMPLE 20 *For each stock currently rising in price, when did it start rising?*

```
retrieve (S.Stock)
valid at begin of rising(S.Price by S.Stock)
```

Stock	*At*
IBM	12-29-91 9:01AM

The adverb "currently" is taken care of with the default *when* clause, in this case, "`when S overlap present`". □

To get the history of the rising stocks, we simply substitute another *when* clause.

EXAMPLE 21 *When was each stock's price rising?*

```
retrieve (S.Stock)
valid during rising(S.Price by S.Stock)
when true
```

Stock	*From*	*To*
IBM	6-3-91 11:23AM	10-30-91 4:57PM
IBM	10-30-91 4:57PM	12-3-91 8:44AM
IBM	12-29-91 9:01AM	forever

The price is rising until the moment it decreases in value. □

6.2.7 Valid-time Indeterminacy

Often facts are not known to within the accuracy of the time granularity of the DBMS, which might be a second or even a microsecond [Dyreson & Snodgrass 1992A].

EXAMPLE 22 *The times for buy and sell orders are known only within a three hour interval.*

```
modify Owns to indeterminate span = %3 hours%
```

A buy order received at 7:30AM is recorded at 6AM with a 3 hour indeterminacy span (from 6AM to 9AM). We specify this at the schema level; indeterminacy spans can also be indicated at the per-tuple, extensional level. While we can also associate a probability distribution function with that indeterminate span, we assume the default, the uniform distribution. □

EXAMPLE 23 *What stocks did Melanie definitely own at 1AM this morning?*

```
retrieve (O.all)
valid at |1AM|
where O.Client = "Melanie"
when O overlap |1AM|
```

The default is only to retrieve tuples that fully satisfy the predicate. This is consistent with the Quel semantics. □

Valid-time indeterminacy enters queries at two places, specifying the *range credibility* of the underlying information to be utilized in the query, and specifying the *ordering plausibility* of temporal relationships expressed in the when and valid clauses. We illustrate only ordering plausibility here.

EXAMPLE 24 *What stocks did Melanie probably own?*

```
retrieve (O.all)
valid at |1AM|
where O.Client = "Melanie"
when O overlap |1AM| with plausibility 70
```

Here, "probably" is specified as a plausibility of 70%. We'll give the formal semantics for this query in Section 6.3.6, and an algebraic version in Section 6.4.6. □

EXAMPLE 25 *What stocks did Melanie perhaps own?*

```
retrieve (O.all)
valid at |1AM|
where O.Client = "Melanie"
when O overlap |1AM| with plausibility 30
```

We associate "perhaps" with a plausibility of 30%. □

EXAMPLE 26 *What stocks might Melanie possibly have owned at 1AM?*

```
set default plausibility = 1

retrieve (O.all)
valid at |1AM|
where O.Client = "Melanie"
when O overlap |1AM|
```

A plausibility of 1% allows any overlap that was even remotely possible to satisfy the when clause. □

6.2.8 Update Statements

Quel has three update statements, `append`, `delete`, and `replace`.

EXAMPLE 27 *On July 15, 1992, at 3PM, Melanie bought 20 shares of DEC stock.*

```
append to Own(Client="Melanie", Stock="DEC", Shares=20)
valid during [3PM July 15, 1992, forever]
```

The "`[...]`" is an *interval constant* [Soo & Snodgrass 1992B]. Here we assume that Melanie doesn't yet have any DEC stock. This buy order was executed, probably by a batch program driven by stock exchange information, at 11PM. A query of Melanie's stocks executed before 11PM would not include this stock. The algebraic equivalent of this update is given in Section 6.4.5. □

There is no `as of` clause in any of the update statements, and no specification of a new transaction time. The transaction time is when the append statement was executed, and is supplied by the system.

EXAMPLE 28 *Actually, an error was made: the request came in at noon.*

```
replace O("Melanie", "DEC", O.Shares)
valid during [12PM July 15, 1992, 3PM July 15, 1992]
where O.Client = "Melanie" and O.Stock = "DEC"
when O overlap |3PM July 15, 1992|
```

In this modification statement, we update the number of stocks owned between noon and 3PM to the value valid at 3PM. □

EXAMPLE 29 *At 9AM on August 20, 1992, an order is received to sell all shares of DEC stock.*

```
delete Own(O.all)
valid during [9AM Aug 20, 1992, forever]
where O.Client = "Melanie" and O.Stock = "DEC"
```

This transaction was executed at 1PM, and the change was recorded in the database at that time. □

6.2.9 Schema Evolution

Often the database schema needs to be modified to accommodate new applications. The ***modify*** statement has several variants, allowing any previous decision to be later changed or undone.

One use of the modify statement is to specify primary storage structures and secondary indexes. There are a variety of possible storage structures available. One promising approach, the *temporally partitioned store*, divides the data into the *current store*, containing the current data and possibly some history data, and the *history store*, holding the rest of the data [Ahn & Snodgrass 1988]. The two stores can utilize different storage formats, and even different storage media, depending on the individual data characteristics. We have cataloged several formats for the history store, including reverse chaining, accession lists, clustering, stacking, and cellular chaining. The last, cellular chaining, can be regarded as a combination of reverse chaining and stacking, in that it links together tuples with identical values of one or more domains, forming a history of some object or relationship. It also has the benefit of physical clustering.

EXAMPLE 30 *Use cellular clustering as the primary storage structure for the* `Stocks` *relation.*

```
modify Stocks to cellular on Stock where cellsize = 15
```

Here, the cellsize is the number of tuples to cluster on a page. □

We can also specify secondary indexes, which can optionally incorporate valid and transaction timestamps (making them more useful in processing when clauses, but also increasing their size).

EXAMPLE 31 *Add a secondary index on the* `Stock` *attribute of the* `Own` *relation.*

```
modify Own to index on Stock as persistent historical
```

The index just specified will include both the valid and transaction timestamps. The keyword "***historical***" was assigned before this term was refined to the more precise "valid-time". □

The query evaluation performance can be greatly improved through the use of appropriate storage structures and indexes. In fact, without them, performance is uniformly discouraging [Ahn & Snodgrass 1986]. To analyze the performance of temporal queries on databases using various access methods, we have developed an analytical model that takes a temporal query and a database schema as input, and outputs the estimated I/O cost for that query on that database [Ahn 1986, Ahn & Snodgrass 1989]. This model has been validated with measurements obtained from a prototype implementation.

The modify statement can also be used to change the attributes associated with a relation.

EXAMPLE 32 *Add an attribute to* `Stocks` *that records the number of shares traded.*

```
modify Stocks (Stock=S.Stock, Price=S.Price, NumTraded:Integer=0)
```

We need to specify a value for a new column, in this case 0. □

Schema evolution involves transaction time, as it concerns how the data is stored in the database [McKenzie 1988, McKenzie & Snodgrass 1990]. As an example, changing the type of a relation from a valid-time relation to a bitemporal relation will cause future intermediate states to be recorded; states stored when the relation was a valid-time relation are not available.

EXAMPLE 33 *The* `Stocks` *relation should also record all errors.*

```
modify Stocks to persistent
```

This schema modification was executed on September 3, 1992. We can now rollback to states after that date. □

Still later, we no longer require the `Stocks` relation.

EXAMPLE 34 *Remove the* `Stocks` *relation.*

```
destroy Stocks
```

This schema modification was executed on October 17, 1992. We can still rollback to states between September 3, 1992, when transaction time was supported for this relation, and October 17, 1992; we cannot access the relation at all after that later date. □

6.3 Formal Semantics

A formal tuple calculus semantics exists for the entire language. In this section we introduce the tuple calculus, discuss the semantics of the basic retrieve statement, then consider the more involved aspects of aggregation, valid-time indeterminacy, and update. We end this section by discussing reducibility to the Quel semantics.

6.3.1 The Tuple Calculus

Tuple relational calculus statements are of the form

$$\{u^i \mid \psi(u)\} \tag{6.1}$$

where the variable t denotes a tuple of arity i, and $\psi(t)$ is a first order predicate calculus expression containing only one free tuple variable t. $\psi(t)$ defines the tuples contained in the relation specified by the Quel statement. The tuple calculus statement for the skeletal Quel statement

> ***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

is

$$\begin{aligned}\{u^r \mid (\exists t_1)\ldots(\exists t_k)(\, & R_1(t_1) \wedge \ldots \wedge R_k(t_k) \\ & \wedge u[1] = t_{i_1}[j_1] \wedge \ldots \wedge u[r] = t_{i_r}[j_r] \\ & \wedge \psi')\},\end{aligned} \tag{6.2}$$

which states that each t_i is in R_i, that each result tuple u is composed of r components, that the m^{th} attribute of u is equal to the ${j_m}^{th}$ attribute (having an attribute name of D_m) of the tuple variable t_{i_m}, and that the condition ψ' (ψ trivially modified for attribute names and Quel syntax conventions) holds for u [Ullman 1988]. The first line corresponds to the relevant range statements, the second to the target list, and the third to the where clause. The skeletal Quel statement is not completely general, since attribute names for the derived relation must be provided in the target list, and attribute values may be expressions. We ignore such details for the remainder of the chapter.

The semantics of a query on a temporal database will be specified by providing a tuple calculus statement that denotes a snapshot relation embedding a bitemporal relation which is the result of the query. This snapshot relation has as its schema four additional *explict* attributes, all timestamps: valid from, valid to, transaction start, and transaction stop. The tuple calculus statement for a TQuel retrieve statement is very similar to that of a Quel retrieve statement: additional components corresponding to the valid, when, and as-of clauses are also present. Although the expressions appearing in all three clauses are similar syntactically, their semantics are rather different.

6.3.2 Temporal Constructors

The valid clause specifies the time during which the derived tuple is valid. A temporal constructor is used to specify a time value. The time value returned by this expression will in fact be one of the time values contained in one of the tuples associated with the variables involved in that expression. Hence, the expression is not actually *deriving* a *new* time value from the given time values; rather, it is *selecting* one of the *given* time values. Of course, the selection criterion can, and indeed usually does, depend on the relative temporal ordering of the original events.

The approach taken here associates each temporal constructor with a function on one or two intervals, returning an interval (events are represented as intervals with identical begin and end timestamps). Tuple variables are replaced with their associated valid time values. The result of an expression of an event type will hence be one of these time values. Individual time values are denoted with a *chronon* number, represented in the database as a 8- to 12-byte structure [Dyreson & Snodgrass 1992B]. The granularity of time (e.g., nanosecond, month, year) is fixed by the DBMS. Note that when we speak of a "point in time," we actually refer to a chronon, which is an interval whose duration is determined by the granularity of the measure of time being used to specify that point in time [Anderson 1982].

We define the temporal constructors after first defining a few auxiliary functions on timestamps (*First*, *Last*) or tuple variables (*event*, *interval*).

$$\mathit{First}(\alpha,\beta) \triangleq \begin{cases} \alpha & \text{if } \mathit{Before}(\alpha,\beta) \\ \beta & \text{otherwise} \end{cases} \tag{6.3}$$

$$\mathit{Last}(\alpha,\beta) \triangleq \begin{cases} \beta & \text{if } \mathit{Before}(\alpha,\beta) \\ \alpha & \text{otherwise} \end{cases} \tag{6.4}$$

$$\mathit{event}(t) \triangleq \langle t[\mathit{at}], t[\mathit{at}] \rangle \tag{6.5}$$

$$\mathit{interval}(t) \triangleq \langle t[\mathit{from}], t[\mathit{to}] \rangle \tag{6.6}$$

$$\mathit{beginof}(\langle \alpha,\beta \rangle) \triangleq \langle \alpha,\alpha \rangle \tag{6.7}$$

$$\mathit{endof}(\langle \alpha,\beta \rangle) \triangleq \langle \beta,\beta \rangle \tag{6.8}$$

$$\mathit{overlap}(\langle \alpha,\beta \rangle, \langle \gamma,\delta \rangle) \triangleq \langle \mathit{Last}(\alpha,\gamma), \mathit{First}(\beta,\delta) \rangle \tag{6.9}$$

$$\mathit{extend}(\langle \alpha,\beta \rangle, \langle \gamma,\delta \rangle) \triangleq \langle \mathit{First}(\alpha,\gamma), \mathit{Last}(\beta,\delta) \rangle \tag{6.10}$$

A few comments are in order. First, these functions all apply to one or more *pairs* of timestamps, denoted "$\langle\ \rangle$", and return a timestamp pair. If the expression is of type event, then the denotation of the expression will be defined to be the time value appearing as the first element of the ordered pair resulting from the application of these functions on the underlying tuples. The definitions ensure that the first element will be identical to the second element. Secondly, while the *Before* predicate is simply "$\leq$" on timestamps, we retain this predicate because it will be generalized when valid-time indeterminacy is considered. Third, the translation is *syntax-directed*: the semantic functions are in correspondence with the productions of the grammar (given in the Appendix) for e-expressions [Ceri & Gottlob 1985]. And finally, the definition of the *overlap* function assumes that the intervals do indeed overlap; if this constraint is satisfied, then the ordered pairs $\langle \alpha,\beta \rangle$ generated by these functions will always represent intervals, i.e., the ordered pairs will satisfy $\mathit{Before}(\alpha,\beta)$. Invalid temporal constructors will be handled with an additional clause in the tuple calculus statement to be presented shortly.

The temporal constructors appearing in the as-of clause can be replaced with their functions on ordered pairs of timestamps and the temporal constants (strings) can be replaced by their corresponding ordered pairs of timestamps. The resulting expression can be evaluated at compile-time, resulting in a single event or interval.

6.3.3 Temporal Predicates

The when clause is the temporal analogue of the where clause. The temporal predicate in the when clause determines whether the tuples may participate in the computation by examining their timestamp attributes. Expressing this formally involves generating a conventional predicate on the timestamp attributes of the argument relations. Such predicates are generated in three steps. First, the tuple variables and the temporal constructors are replaced by the functions defined in the previous subsection. Second, the `and`, `or`, and `not` operators are replaced by the logical predicates. Finally, the temporal predicate operators (`precede`, `overlap` and `equal`) are replaced by the following predicates on ordered pairs of timestamps.

$$precede(\langle\alpha,\beta\rangle,\langle\gamma,\delta\rangle) \triangleq \mathit{Before}(\beta,\gamma) \tag{6.11}$$

$$overlap(\langle\alpha,\beta\rangle,\langle\gamma,\delta\rangle) \triangleq \mathit{Before}(\alpha,\delta)\wedge\mathit{Before}(\gamma,\beta) \tag{6.12}$$

$$\begin{aligned} equal(\langle\alpha,\beta\rangle,\langle\gamma,\delta\rangle) \triangleq\ & \mathit{Before}(\alpha,\gamma)\wedge\mathit{Before}(\gamma,\alpha)\\ & \wedge\mathit{Before}(\beta,\delta)\wedge\mathit{Before}(\delta,\beta) \end{aligned} \tag{6.13}$$

The result is a conventional predicate on the valid times of the tuple variables appearing in the when clause.

EXAMPLE 35 Applying the first step to the following temporal predicate, used in a query in Example 12 in Section 6.2.4,

```
(end of S + %1 month%) overlap S2
        and O overlap end of S and O overlap begin of S2,
```

is translated into the following steps.

$$\begin{aligned} & (sum(endof(interval(S)),\ \texttt{\%1 month\%}))\ \texttt{overlap}\ interval(S2)\\ & \quad \texttt{and}\ interval(O)\ \texttt{overlap}\ endof(interval(S))\\ & \quad\quad \texttt{and}\ interval(O)\ \texttt{overlap}\ beginof(interval(S2))\\ \to\ & (sum(endof(\langle S[from],\ S[to]\rangle),\ \texttt{\%1 month\%}))\ \texttt{overlap}\ \langle S2[from],\ S2[to]\rangle\\ & \quad \texttt{and}\ \langle O[from],\ O[to]\rangle\ \texttt{overlap}\ endof(\langle S[from],\ S[to]\rangle)\\ & \quad\quad \texttt{and}\ \langle O[from],\ O[to]\rangle\ \texttt{overlap}\ beginof(\langle S2[from],\ S2[to]\rangle)\\ \to\ & (sum(S[to],\ \texttt{\%1 month\%}))\ \texttt{overlap}\ \langle S2[from],\ S2[to]\rangle\\ & \quad \texttt{and}\ \langle O[from],\ O[to]\rangle\ \texttt{overlap}\ \langle S[to],\ S[to]\rangle\\ & \quad\quad \texttt{and}\ \langle O[from],\ O[to]\rangle\ \texttt{overlap}\ \langle S2[from],\ S2[from]\rangle \end{aligned}$$

The second step results in

$$\begin{aligned} \to\ & sum(S[to],\ \texttt{\%1 month\%})\ \texttt{overlap}\ \langle S2[from],\ S2[to]\rangle\\ & \quad \wedge\langle O[from],\ O[to]\rangle\ \texttt{overlap}\ \langle S[to],\ S[to]\rangle\\ & \quad\quad \wedge\langle O[from],\ O[to]\rangle\ \texttt{overlap}\ \langle S2[from],\ S2[from]\rangle \end{aligned}$$

and the third step results in

$$\begin{aligned} \to\ & \mathit{Before}(sum(S[to],\ \texttt{\%1 month\%}),\ S2[to])\\ & \quad\quad \wedge\mathit{Before}(S2[from],\ sum(S[to],\ \texttt{\%1 month\%}))\\ & \quad \wedge\mathit{Before}(O[from],\ S[to])\wedge\mathit{Before}(S[from],\ O[to])\\ & \quad\quad \wedge\mathit{Before}(O[from],\ S2[to])\wedge\mathit{Before}(S2[from],\ O[to]). \qquad \square \end{aligned}$$

This transformation process always results in a predicate that mentions only the functions *First*, *Last*, and *Before*.

6.3.4 The Retrieve Statement

A formal semantics for the TQuel retrieve statement can now be specified. Let Φ_ϵ be the function corresponding to the e-expression ϵ with the operators replaced with logical predicates. Let Γ_τ be the predicate corresponding to the temporal predicate τ as generated by the process discussed in Section 6.3.3. Note that Φ_ϵ and Γ_τ will contain only the functions *First* and *Last* and the predicates *Before*, $\wedge$, $\vee$, $\neg$; the rest of the functions, and Φ_α entirely (where α appears in an as-of clause), can be evaluated at compile-time. Of course, the defaults provide the appropriate expressions when a clause is not present in the query. Given the query

$$\begin{array}{l} \texttt{range of } t_1 \texttt{ is } R_1 \\ \ldots \\ \texttt{range of } t_k \texttt{ is } R_k \\ \texttt{retrieve } (t_{i_1}.D_{j_1}, \ldots, t_{i_r}.D_{j_r}) \\ \texttt{valid during } \upsilon \\ \texttt{where } \psi \\ \texttt{when } \tau \\ \texttt{as of } \alpha \end{array} \tag{6.14}$$

the tuple calculus statement has the following form.

$$\begin{array}{ll} \{u^{(r+2)} & | \ (\exists t_1)\ldots(\exists t_k)(R_1(t_1) \wedge \ldots \wedge R_k(t_k)) \\ & \wedge u[1] = t_{i_1}[j_1] \wedge \ldots \wedge u[r] = t_{i_r}[j_r] \\ & \wedge u[r+1] = \mathit{beginof}(\Phi_\upsilon) \wedge u[r+2] = \mathit{endof}(\Phi_\upsilon) \\ & \wedge \mathit{Before}(u[r+1], u[r+2]) \\ & \wedge \psi' \\ & \wedge \Gamma_\tau \\ & \wedge (\forall l)(1 \leq l \leq k.(\mathit{overlap}(\Phi_\alpha, \langle t_l[start], t_l[stop] \rangle))) \\ &)\} \end{array} \tag{6.15}$$

The first line states that each tuple variable ranges over the correct relation, and is from the Quel semantics. The resulting tuple is of arity $r+2$, and consists of r explicit attributes and two implicit attributes ($from$ and to). The second line, also from the Quel semantics, states the origin of the values in the explicit attributes of the derived relation. The third line originates in the valid clause, and specifies the values of the $from$ and to valid times. Notice that these times must obey the specified ordering. The fourth line ensures that a legitimate interval results. The next line originates in the where clause, and is from the Quel semantics. The sixth line is the predicate from the when clause. The last line originates in the as-of clause, and states that the tuple associated with each tuple variable must have a transaction interval that overlaps the interval specified in the as-of clause (Φ_α will be a constant time value, i.e., a specific timestamp or pair of timestamps).

Note that Γ_τ and Φ_υ are functions over the *from* and *to* attributes of a subset of the tuple variables. If t is a tuple variable associated with an interval relation and appears in a temporal constructor or predicate, then the *from* and *to* time values are passed to the relevant function; if t is associated with an event relation, then only the *at* time value is used.

EXAMPLE 36 The query in Example 14 in Section 6.2.5, printing the current worth of Melanie's stocks, has the following semantics.

$$
\begin{aligned}
\{u^4 \mid\ & (\exists O)(\exists S)(\mathit{Stocks}(S) \wedge \mathit{Own}(O) \\
& \wedge u[1] = O[Stock] \wedge u[2] = O[Shares] * S[Price] \\
& \wedge u[3] = Last(O[\mathit{from},\ S[\mathit{from}]) \wedge u[4] = \mathit{First}(O[\mathit{to}],\ S[\mathit{to}]) \\
& \wedge \mathit{Before}(u[3], u[4]) \\
& \wedge O[Client] = \texttt{"Melanie"} \wedge O[Stock] = S[Stock] \\
& \wedge \mathit{Before}(Last(O[\mathit{from}],\ S[\mathit{from}]),\ now) \wedge \mathit{Before}(now, \mathit{First}(O[\mathit{to}],\ S[\mathit{to}])) \\
& \wedge \mathit{overlap}(now, \langle O[\mathit{start}],\ O[\mathit{stop}]\rangle) \wedge \mathit{overlap}(now, \langle S[\mathit{start}],\ S[\mathit{stop}]\rangle) \\
&)\}
\end{aligned}
$$

□

Note that the semantics of the TQuel specific constructs is quite similar to that of their Quel counterparts.

6.3.5 Aggregates

Our approach to the semantics is based on Klug's method, which was used in a separate, more formal tuple relational calculus [Klug 1982]. 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, rather than to a set of values drawn from a single attribute's domain, the distinction between unique and non-unique aggregation can be preserved.

The values of TQuel aggregates change 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 clause in Quel: they determine which tuples from the underlying relations participate in the remainder of the query. These selected tuples are combined with the values computed in the aggregate sets to obtain the final output relation.

Aggregates always generate temporary interval relations, even though an aggregated attribute can appear in a query that results in an event relation. This temporary relation has exactly one value at any point in time (for an aggregate with a by clause, 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 *time-partition* of a set of relations $R_1, \ldots, R_k$, relative to a given window function w, to be defined shortly, as

$$
\begin{aligned}
T(R_1,\, \ldots,\, R_k,\, w) \triangleq\ & \{0,\ \text{forever}\} \\
& \cup \{s \mid (\exists x)(\exists i) \qquad\qquad\qquad\qquad\qquad\qquad (6.16)\\
& \quad (1 \leq i \leq k \wedge R_i(x) \wedge (s = x[from] \vee s = x[to] \\
& \quad\quad \vee (\exists t)(s = t \wedge t - w(t) = x[to] \\
& \quad\quad\quad \wedge \forall t',\, t' > t,\, t' - w(t') > x[to])))\}\,.
\end{aligned}
$$

The time-partition brings together all the times (chronons) 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. w maps each time t into its aggregation window size.

EXAMPLE 37 The clause "`for each %month%`", given in the query in Example 19 in Section 6.2.6, implies a window size dependent on the timestamp granularity. Let us assume an underlying granularity of a day. Then the window function for this example 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 20, 1980, the first day in the aggregate window, was 27 days before March 20). □

If two times y and z are consecutive in the set 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

$$
\begin{aligned}
& Constant(R_1,\, \ldots,\, R_k,\, w) \qquad\qquad\qquad\qquad\qquad\qquad (6.17)\\
& \triangleq \{\langle y,\, z\rangle \mid y \in T(R_1,\, \ldots,\, R_k,\, w) \wedge z \in T(R_1,\, \ldots,\, R_k,\, w) \\
& \quad \wedge y \neq z \wedge \mathit{Before}\ (y,\ z) \\
& \quad \wedge (\forall e)(e \in T(R_1,\, \ldots,\, R_k,\, w) \Rightarrow (\mathit{Before}(e,\ y) \\
& \quad\quad \vee Equal(e,\, y) \vee \mathit{Before}(z,\ e) \\
& \quad\quad \vee Equal(z,\, e)) \\
& \}.
\end{aligned}
$$

The last three 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 $\langle y,\, z\rangle$ 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 $\langle y,\, z\rangle$ 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*.

In general, a partition is defined for each aggregate, on which the aggregate function is applied.

EXAMPLE 38 We illustrate by giving the semantics of the query in Example 19 in Section 6.2.6 selecting those stocks that have doubled over the last month. The partition is indexed by the `Stock` attribute, as well as an interval in the *Constant* set (which in this case simply returns the intervals in the `Stocks` relation).

$$P(Stock,\ y,\ z) = \{u^1 \mid (\ Stocks(u) \wedge u[1] = Stock \\ \wedge overlap(\langle y,\ z\rangle,\ \langle u[from],\ u[to] + w'(y)\rangle)\}$$

The window function w' in the second line corresponds to "`for each %month%`" in the retrieve statement. This line indicates that all tuples participating in the aggregate must overlap the interval $\langle y,\ z\rangle$. From the definition of the $Constant$ interval set, which supplies the intervals $\langle y,\ z\rangle$, 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 w' is used in a similar fashion to the definition of the time partition. Should the aggregate contain a where, when, or as of clause, these clauses would have been accommodated in this partition.

The output relation from the query is

$$\begin{aligned}\{u^{(2+2)} \mid\ & (\exists S)(\exists y)(\exists z)(Stocks\,(S) \\ & \wedge\langle y,\ z\rangle \in Constant(Stocks,\ w') \wedge overlap(\langle y,\ z\rangle,\ \langle S[from],\ S[to]\rangle) \\ & \wedge u[1] = S[Stock] \wedge u[2] = S[Price] \\ & \wedge u[3] = rate(P(u[Stock],\ y,\ z))[2] \\ & \wedge u[4] = last(y,\ S[from]) \wedge u[5] = First(z,\ S[to]) \wedge Before(u[4],\ u[5]) \\ & \wedge overlap(now,\ \langle S[from],\ S[to]\rangle) \\ &)\}\,.\end{aligned}$$

A comparison with the tuple calculus expression for the TQuel retrieve statement given earlier reveals that lines two and four are new and lines one and five are altered. In line two, the $Constant$ interval set provides the interval $\langle y,\ z\rangle$ 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 w' appears explicitly as an argument to the $Constant$ interval set and implicitly in P. Line two also ensures that the intervals associated with the tuple variables aggregated over as well as with those tuple variables specified in the by-clause overlap with the interval during which the aggregate is constant. Line four computes the aggregate. The $rate$ function that is applied to each partition is defined elsewhere [Snodgrass, et al. 1992]. Line five ensures that the valid time of the result relation is the intersection with the specified valid time and the interval $\langle y,\ z\rangle$). □

6.3.6 Valid-time Indeterminacy

The changes to the semantics to support valid-time indeterminacy are quite minimal. There are two aspects that require support. *Range credibility* restricts the range of the indeterminate events participating in the query. Effectively, non-credible starting and terminating times are eliminated to the chosen level of credibility during query processing, allowing the user to control the quality of the information used in the query. *Ordering plausibility*, exemplified in Section 6.2.7, controls the construction of an answer to the query using the pool of credible information. It allows the user to express a level of plausibility in a temporal relationship such as `precede`.

To permit timestamps to model indeterminate events, we incorporate in timestamps an indeterminacy span, indicating the period of time when the event *could* have occurred, and a probability distribution, indicating the probability of the event occurring before or during each chronon in the indeterminacy span. In the example in Section 6.2.7, both are specified at the schema level, and hence need not be modeled explicitly in the timestamps. However, in the general case, both need to be efficiently represented in an individual timestamp [Dyreson & Snodgrass 1992A, Dyreson & Snodgrass 1992B].

To support range credibility, two functions are introduced. These functions compute a "shortened" version of an indeterminate event by shrinking its set of possible events and modifying its probability distribution [Dyreson & Snodgrass 1992A].

To support ordering plausibility we redefine the ordering relation ***Before***. The semantics of retrieve without indeterminacy given in the preceding sections is based on a well-defined ordering of the valid time events in the underlying relations. Every temporal predicate and temporal constructor refers to this ordering to determine if the predicate is true or the constructor succeeds. A set of determinate events has a single temporal ordering. Given a temporal expression consisting of temporal predicates and temporal constructors, this ordering either satisfies the expression or fails to satisfy it.

A set of indeterminate events, however, typically has many possible temporal orderings. Some of these temporal orderings are plausible while others are implausible. The user specifies which orderings are plausible by setting an appropriate ordering plausibility value. We stipulate that a temporal expression is satisfied if there exists a plausible ordering that satisfies it.

The temporal ordering is given by the ***Before*** relation. In the determinate semantics, ***Before*** is the "$\leq$" relation on event times. In the indeterminate semantics, the temporal ordering depends on a probabilistic ordering operator "$\leq_{prob}$" which is defined as follows. For any two independent indeterminate events, $\alpha = ([\alpha_1,\ \alpha_m],\ P_\alpha)$ and $\beta = ([\beta_1,\ \beta_n],\ P_\beta)$,

$$\alpha \leq_{prob} \beta = \lfloor 100 \times (\sum_{E_i=\alpha_1}^{\alpha_m} \sum_{E_j=\beta_1}^{\beta_n} (if\ \ E_i \leq E_j\ \ then\ \ P_\alpha(E_i) \times P_\beta(E_j)\ \ else\ \ 0))\rfloor, \tag{6.18}$$

where the possible chronons are ordered by the "$\leq$" operator on the integers.

We modify ***Before*** to include an additional initial parameter, the ordering plausibility γ. The value of γ can be any integer between 1 and 100 (inclusive). In general, higher (closer to 100) ordering plausibilities stipulate that fewer orderings should be considered plausible. The indeterminate $Before$ is defined as follows.

$$\textit{\textbf{Before}}(\gamma,\ \alpha,\ \beta) = \begin{cases} TRUE & (\alpha\ is\ \beta) \vee ((\alpha \leq_{prob} \beta) \geq \gamma) \\ FALSE & \text{otherwise} \end{cases} \tag{6.19}$$

An event is defined to be *Before* itself, for all values of γ. Two events are said to be ***equivalent*** if they have both the same set of possible chronons and the same probability distribution. Two equivalent, but not identical, events may or may not be *Before* one another, depending on γ.

EXAMPLE 39 With this apparatus, we can give the semantics for the query given in Example 24 in Section 6.2.7, listing the stocks that Melanie probably owned at 1AM.

$$\begin{aligned}\{u^{(5)} \mid (\exists O)(\ & Own(O) \\ & \wedge u[1] = O[Stock] \wedge u[2] = O[Stock] \wedge u[3] = O[Shares] \\ & \wedge u[4] = \texttt{|1AM|} \\ & \wedge O[Client] = \texttt{"Melanie"} \\ & \wedge \mathit{Before}(70,\ O[\mathit{from}],\ \texttt{|1AM|}) \wedge \mathit{Before}(70,\ \texttt{|1AM|},\ O[\mathit{to}]) \\ & \wedge \mathit{overlap}(present, \langle O[\mathit{start}],\ O[\mathit{stop}]\rangle) \\ &)\}\end{aligned}$$

□

Elsewhere we give the semantics for the generic TQuel retrieve statement, incorporating both range credibility and order plausibility [Dyreson & Snodgrass 1992A].

6.3.7 The Update Statements

In examining the semantics of the TQuel update, we proceed by first considering the skeletal Quel append statement,

$$\texttt{append to}\ R\ (t_{i_1}.D_{j_1},\ \ldots,\ t_{i_r}.D_{j_r})$$
$$\texttt{where}\ \psi,$$

which has the tuple calculus semantics

$$\begin{aligned}R' = R \cup \{u^r \mid (\exists t_1)\ldots(\exists t_k)(\ & R_1(t_1) \wedge \ldots \wedge R_k(t_k) \\ & \wedge u[1] = t_{i_1}[j_1] \wedge \ldots \wedge u[r] = t_{i_r}[j_r] \\ & \wedge \psi')\}.\end{aligned} \tag{6.20}$$

The set being appended is identical to that for the Quel retrieve statement (see Section 6.3.1). Note that the set being appended may contain tuples already in R.

The semantics for the TQuel append statement is somewhat complicated, because the set to be unioned with the existing relation should only contain tuples that are not valid in the existing relation. We cannot depend on the union working correctly when the tuples being appended are identical to tuples in the current valid-time relation. So the semantics only appends tuples during those times when a tuple with identical explicit attributes is *not* valid. We do not give the full semantics here; it is available elsewhere [Snodgrass 1987].

The tuple calculus semantics of the delete statement shows a similar increase in complexity. This statement will perhaps change some transaction stop times from until changed to now, logically removing them, and will perhaps also add tuples with a transaction start time of now, for those portions of time *not* logically deleted. Hence, the semantics for logical deletion is physical insertion!

The semantics of the replace statement is even more complex. The replace statement has a semantics similar to that of a delete statement followed by an append statement. It is not equivalent to a delete followed by an append when the expressions in the target list mention the primary tuple variable. Hence, the semantics must be careful to union just the right amount of information implied by logical deletion, followed by just the right amount of information implied by the append.

6.3.8 Reducibility

If a TQuel statement does not contain a valid, when, or as-of clause, then it looks identical to a standard Quel retrieve statement; thus it should have an identical semantics. However, an Ingres database is not temporal; it is a snapshot database. Hence, the tuples participating in a Quel statement are in the snapshot relation that is the result of the last transaction performed on the database (i.e., are *current*) and are valid at the time the statement is executed. This *snapshot database slice* (all current tuples valid at a particular time τ) is formed by first eliminating the event relations (since snapshot relations cannot represent events at all), eliminating all tuples with a $start$ time greater than τ and with a $stop$ time less than τ, eliminating all tuples not valid at τ, and finally removing the implicit time attributes. Then reducibility is satisfied if taking a snapshot of the result of applying a query with the TQuel semantics just specified is identical to the result of applying the same query with the Quel semantics to a snapshot of the database.

Theorem 1 *The TQuel semantics reduces to the standard Quel semantics when applied to a snapshot database slice of the temporal database.*

The proof of this equality revolves around the defaults for the valid, when, and as-of clauses [Snodgrass 1987]. The defaults effectively take a database slice at $\tau = now$, which is the time the query is executed. The default when and valid clauses state that all the underlying tuples are valid for the entire interval the resulting tuple was valid. The resulting tuples are guaranteed to be current by the tuple calculus semantics of the retrieve statement.

A similar reduction can be argued concerning queries with aggregates and with valid-time indeterminacy, and concerning the update statements, as the defaults were specifically chosen to ensure their reducibility to the standard Quel semantics. The benefit of these reductions is that the intuition and understanding gained by using Quel on a snapshot database applies to TQuel on a temporal database.

6.3.9 Summary

In this section, we first presented the tuple calculus semantics of Quel, then proceeded to extend this semantics. We added support for temporal constructors, used in the valid clause, temporal predicates, used in the when clause, aggregates, which required an auxiliary partition and a constant interval set for each aggregate, and valid-time indeterminacy, which required augmenting all the temporal predicates and constructors with another parameter, the order plausibility. The update statements were transformed into set unions. Finally, we argued that this semantics is a faithful extension of the Quel semantics.

6.4 A Temporal Algebra

We now extend the relational algebra [Codd 1970] to enable it to handle time. Several benefits accrue from defining a temporal algebra. A temporal algebra is useful in the formulation of a temporal data model because it defines formally the types of objects and the operations on object instances allowed in the data model. Similarly, the algebra provides support of calculus-based query languages. Also, implementation issues, such as query optimization and physical storage strategies, can best be addressed in terms of the algebra.

The relational algebra already supports user-defined time in that user-defined time is simply another domain, such as integer or character string, provided by the DBMS [Bontempo 1983, Overmyer & Stonebraker 1982, Tandem 1983]. The relational algebra, however, supports neither valid time nor transaction time. Hence, for clarity, we refer to the relational algebra hereafter as the *snapshot* algebra and our proposed algebra, which supports valid time, as a *valid-time* algebra. We later embed the algebra in a language to support transaction time. We provide formal definitions for valid-time relations and for their associated algebraic operators. The result is an algebraic language supporting all three kinds of time.

In this section we define the valid-time algebra and provide a formal semantics. To do so, we redefine a *relation*, the only type of object manipulated in the algebra, to include valid time. We also redefine the existing relational algebraic operators, and introduce new operators, to handle this new temporal dimension. We then show that the algebra has the expressive power of TQuel facilities that support valid time. We demonstrate that several important properties hold: closure, relational completeness, and a restricted form of reducibility.

6.4.1 Data Model

The algebra presented here is an extension of the snapshot algebra. As such, it retains the basic restrictions on attribute values found in the snapshot algebra. Neither set-valued attributes nor tuples with duplicate attribute values are allowed. Valid time is represented by a set-valued timestamp that is associated with individual attribute values. A timestamp represents possibly disjoint intervals and the timestamps assigned to two attributes in a given tuple need not be identical.

Assume that we are given a relation scheme defined as a finite set of attribute names $\mathcal{N} = \{N_1, \ldots, N_m\}$. Corresponding to each attribute name $N \in \mathcal{N}$ is a domain $Dom(N)$, an arbitrary, non-empty, finite or denumerable set [Maier 83]. Let the positive integers be the domain $\mathcal{T}$, where each element of $\mathcal{T}$ represents a chronon. Also, let the domain $\wp(\mathcal{T})$ be the power set of $\mathcal{T}$. An element of $\wp(\mathcal{T})$ is then a set of integers, each of which represents an interval of unit duration. Also, any group of consecutive integers $t_1, \ldots, t_n$ appearing in an element of $\wp(\mathcal{T})$, together represent the interval $\langle t_1, t_n+1\rangle$. An element of $\wp(\mathcal{T})$, termed a *valid-time element*, is thus a union of intervals.

If we let *value* range over the domain $Dom(N_1) \cup \cdots \cup Dom(N_m)$ and *valid* range over the domain $\wp(\mathcal{T})$, we can define an *valid-time tuple ht* as a mapping from the set of attribute names to the set of ordered pairs (*value*, *valid*), with the following restrictions.

- $\forall a,\ 1 \leq a \leq m\ (value(ht[N_a]) \in Dom(N_a)$ and
- $\exists a,\ 1 \leq a \leq m\ (valid(ht[N_a]) \neq \emptyset)$.

Note that it is possible for all but one attribute to have an empty timestamp.

Two tuples, *ht* and *ht'*, are said to be *value-equivalent* ($ht \equiv ht'$) if and only if $\forall A \in \mathcal{N},\ value(ht[A]) = value(ht'[A])$. An *valid-time relation h* is then defined as a finite set of valid-time tuples, with the restriction that no two tuples in the relation are value-equivalent.

EXAMPLE 40 For this and all later examples, assume that the granularity of time is a minute relative to midnight, January 1, 1970. Hence, 1 represents 12:01AM, 1-1-1970. Rather than express sets of chronons as sets of integers, we'll show the integers in Gregorian. We enclose each attribute value in parentheses and each tuple in angular brackets (i.e., $\langle\ \rangle$). The following is the `Stocks` relation in this new representation.

Stocks = { ⟨(IBM, {6-3-91:11:23AM..forever}), (8, {6-3-91:11:23AM..10-1-91:9:28AM})⟩,
⟨(IBM, {6-3-91:11:23AM..forever}), (10, {10-1-91:9:30AM..10-2-91:2:14PM})⟩,
⟨(IBM, {6-3-91:11:23AM..forever}), (12, {10-2-91:2:15PM..10-7-91:10:06AM})⟩,
⟨(IBM, {6-3-91:11:23AM..forever}), (16, {10-7-91:10:07AM..10-15-91:4:34PM,
10-30-91:4:57PM..11-2-91:12:52PM})⟩,
⟨(IBM, {6-3-91:11:23AM..forever}), (19, {10-15-91:4:35PM..10-30-91:4:56PM})⟩,
⟨(IBM, {6-3-91:11:23AM..forever}), (25, {11-2-91:12:53PM..11-5-91:2:01PM})⟩,
⟨(IBM, {6-3-91:11:23AM..forever}), (27, {11-5-91:2:02PM..12-3-91:8:43AM})⟩,
⟨(IBM, {6-3-91:11:23AM..forever}), (23, {12-3-91:8:44AM..12-29-91:9:00AM,
1-3-92:4:28PM..forever})⟩,
⟨(IBM, {6-3-91:11:23AM..forever}), (21, {12-29-91:9:01AM..1-3-92:4:27PM})⟩ } □

6.4.2 Extension of Snapshot Operators

Twelve operators complete the definition of the valid-time algebra. Five of these operators, union, difference, Cartesian product, projection, and selection, are analogous to the five operators that define the snapshot algebra for snapshot relations [Ullman 1988]. Each of these five operators on valid-time relations is represented as $\hat{op}$ to distinguish it from its snapshot algebra counterpart *op*.

We use two auxiliary functions in the formal semantics. Both operate over a set of tuples R.

$$\textbf{NotNull}(R) \triangleq \{R \mid \exists r \in R \wedge \exists A \in \mathcal{N}\,(valid(R[A]) \neq \emptyset)\} \qquad (6.21)$$

This function ensures that all tuples in the resulting relation have at least one attribute value which has a non-null timestamp.

$$\begin{aligned}\mathbf{Reduce}(R) \triangleq \{U^n \mid\ & \forall A \in \mathcal{N}\, \exists r \in R(r \equiv u \wedge \forall t \in valid(u[A])(t \in valid(r[A]))) \\ & \wedge \forall r \in R(r \equiv u \Rightarrow \forall A \in \mathcal{N}(valid(r[A]) \subseteq valid(u[A])))\}\end{aligned} \quad (6.22)$$

Reduce expresses the minimal set of value-equivalent tuples, i.e., the set for which there are no such tuples. The first line ensures that no chronons have been manufactured; the second line ensures that all chronons of R are accounted for. If R is a set of tuples over $\mathcal{N}$, then $\mathbf{Reduce}(\mathbf{NotNull}(R))$ is a valid-time relation.

Let Q and R be valid-time relations of m-tuples over the same relation scheme. Then the valid-time union of Q and R, $Q \,\hat{\cup}\, R$, is the set of tuples that are only in Q, are only in R, or are in both relations, with the restriction that each pair of value-equivalent tuples is represented by a single tuple. The timestamp associated with each attribute of this tuple in $Q \,\hat{\cup}\, R$ is the set union of the timestamps of the corresponding attribute in the value-equivalent tuples in Q and R.

$$Q \,\hat{\cup}\, R \triangleq \mathbf{Reduce}(\{u \mid u \in Q \vee u \in R\}) \quad (6.23)$$

The valid-time difference of Q and R, $Q \,\hat{-}\, R$, is the set of all tuples such that the timestamp of each attribute of a tuple in $Q \,\hat{-}\, R$ must equal the set difference of the timestamps of the corresponding attribute in the value-equivalent tuple in Q and the value-equivalent tuple in R.

$$\begin{aligned} Q \,\hat{-}\, R \triangleq\ & \{q^m \mid \exists q \in Q \ \wedge\ \neg(\exists r \in R\ (r \equiv q))\} \\ & \bigcup \mathbf{NotNull}(\{u^m \mid \exists q \in Q\ \exists r \in R\ (u \equiv q \equiv r \\ & \qquad \wedge \forall A \in \mathcal{N}\ (valid(u[A]) = valid(q[A]) - valid(r[A])))\})\end{aligned} \quad (6.24)$$

Now let Q be a valid-time relation of m_1-tuples and R be a valid-time relation of m_2-tuples. Because valid-time relations are attribute-value timestamped, Cartesian product is a particularly simple operator. The valid-time Cartesian product is identical to that for snapshot relations. In the following, "$\circ$" denotes concatenation.

$$Q \,\hat{\times}\, R \triangleq \{q \circ r \mid \exists q \in Q \ \wedge\ \exists r \in R\} \quad (6.25)$$

EXAMPLE 41 Let O be the current valid-time state from `Own`.

O = {⟨(Melanie, {3-8-91:2:10PM..forever}), (IBM, {6-3-91:11:23AM..forever}),
(20, {9-15-91:3:12PM..12-30-91:10:30AM})⟩,
⟨(Melanie, {3-8-91:2:10PM..forever}), (IBM, {6-3-91:11:23AM..forever}),
(30, {12-30-91:10:31AM..forever})⟩ }

$\mathrm{S}_1 = \mathit{Stocks} \hat{\times} \mathrm{O}$
$\quad = \{\langle(\mathrm{IBM},\ \{\text{6-3-91:11:23AM..forever}\}),\ (8,\ \{\text{6-3-91:11:23AM..10-1-91:9:28AM}\}),$
$\qquad (\mathrm{Melanie},\ \{\text{3-8-91:2:10PM..forever}\}),\ (\mathrm{IBM},\ \{\text{6-3-91:11:23AM..forever}\}),$
$\qquad (20,\ \{\text{9-15-91:3:12PM..12-30-91:10:30AM}\})\rangle,$

...

$\quad \langle(\mathrm{IBM},\ \{\text{6-3-91:11:23AM..forever}\}),\ (21,\ \{\text{12-29-91:9:01AM..1-3-92:4:27PM}\})$
$\qquad (\mathrm{Melanie},\ \{\text{3-8-91:2:10PM..forever}\}),\ (\mathrm{IBM},\ \{\text{6-3-91:11:23AM..forever}\}),$
$\qquad (30,\ \{\text{12-30-91:10:31AM..forever}\})\rangle\ \}$

Since *Stocks* contains 9 tuples and O contains 2 tuples, S_1 will contain 18 tuples. □

To define valid-time selection, let R be a valid-time relation of m-tuples. Also, let F be a boolean function involving the attribute names $N_1,\ \ldots,\ N_R$ from R, constants from the domains $Dom(N_1),\ \ldots,\ Dom(N_R)$, the relational operators $<$, $=$, and $>$, and the logical operators $\wedge$, $\vee$, and $\neg$. To evaluate F for a tuple $r \in R$, we substitute the value components of the attributes of r for all occurrences of their corresponding attribute names in F. Then the valid-time selection $\hat{\sigma}_F(\boldsymbol{R})$ is identical to selection in the snapshot algebra: it evaluates to the set of tuples in $\boldsymbol{R}$ for which F is true.

$$\hat{\sigma}_F(\boldsymbol{R}) \triangleq \{r^m \mid r \in \boldsymbol{R} \wedge F(\mathit{value}(r[1]),\ \ldots,\ \mathit{value}(r[m]))\} \qquad (6.26)$$

EXAMPLE 42 *Select those tuples from* S_1 *with a* `O.Client` *of* `"Melanie"` *and* `O.Stock=S.Stock`.

$$\mathrm{S}_2 = \hat{\sigma}_{Client=\texttt{"Melanie"} \wedge O.Stock=S.Stock}(\mathrm{S}_1)$$ □

Let $\boldsymbol{R}$ be a valid-time relation of m-tuples and let $a_1,\ \ldots,\ a_n$ be distinct integers in the range 1 to m. Like the projection operator for snapshot relation, the projection operator for valid-time relations, $\hat{\pi}_{N_{a_1},\ \ldots,\ N_{a_n}}$, retains, for each tuple, the tuple components that correspond to the attribute names $N_{a_1},\ \ldots,\ N_{a_n}$.

$$\hat{\pi}_{N_{a_1},\ \ldots,\ N_{a_n}}(R) \triangleq \mathbf{Reduce}(\mathbf{NotNull}(\{u^n \mid \exists r \in R\, \forall i, 1 \leq i \leq n\, (u[i] = r[a_i])\})) \qquad (6.27)$$

6.4.3 New, Temporal Operators

We now define three new operators that do not have snapshot analogues. The first, derivation, is a new operator that replaces the timestamp of each attribute value in a tuple with a new timestamp, where the new timestamps are computed from the existing timestamps of the tuple's attributes. The second and third, the snapshot ($\widehat{SN}$) and AT operators, convert between valid-time and snapshot relations.

The derivation operator $\hat{\delta}_{G,\ V_1,\ \ldots,\ V_m}(R)$ determines, for a tuple $r \in \boldsymbol{R}$, new timestamps for r's attributes. The derivation operator first determines all possible assignments of *intervals* to attribute names for which the predicate G on timestamps is true. Hence, an occurrence of an attribute name N in G and in V is intended to be a variable, which evaluates to an interval upon tuple substitution. For each assignment of intervals to attribute names for which G is true, the operator evaluates V_a, $1 \leq a \leq m$. The sets of times resulting from the evaluations of V_a are then combined to form a new timestamp for attribute N_a. For notational convenience, we assume that if only one V-function is provided, it applies to all attributes.

EXAMPLE 43 *Extract from S_2 those intervals of* `Price` *(originally from* `Stocks`*) and* `Shares` *(originally from* `Own`*) that overlap each other and now.*

$$
\begin{aligned}
\mathrm{S}_3 = \hat{\delta}_{G,V}(\mathrm{S}_2) \\
=\{\langle(&\text{IBM}, \{\text{1-3-92:4:28PM..forever}\}), (23, \{\text{1-3-92:4:28PM..forever}\}), \\
&(\text{Melanie}, \{\text{1-3-92:4:28PM..forever}\}), (\text{IBM}, \{\text{1-3-92:4:28PM..forever}\}), \\
&(20, \{\text{1-3-92:4:28PM..forever}\})\rangle\ \}
\end{aligned}
$$

where $G \equiv (Price \cap Shares) \cap now \neq \emptyset$ and $V \equiv Price \cap Shares$. Here G determines whether the overlap occurs, and V calculates this overlap, i.e., the interval during which both were valid. This interval is assigned to all of the attributes. □

The derivation operator performs two functions. First, it performs a selection on the valid component of a tuple's attributes. For a tuple r, if G is false when an interval from the valid component of each of r's attributes is substituted for each occurrence of its corresponding attribute name in G, then the temporal information represented by that combination of intervals is not used in the calculation of the new timestamps of the resulting tuples. Secondly, the derivation operator calculates a new timestamp for each attribute N_a of the resulting tuples from those combinations of intervals for which G is true, using V_a. If $V_1, \ldots, V_m$ are all the same function, the tuple is effectively converted from attribute timestamping to tuple timestamping.

The semantics of the derivation operator is defined using an auxiliary function, **Apply**, that selects an interval from the valid-time element of each attribute's timestamp, applies the predicate G to these intervals, and, if G returns true, evaluates the V_i to generate an output interval.

$$
\begin{aligned}
\mathbf{Apply}(G, V_1, \ldots, V_m, R) \\
= \{u \mid \exists r \in R\,(u \equiv r \\
&\wedge \exists I_i \in \mathit{interval}(valid(r[i])), \\
&\ldots \\
&\wedge \exists I_i \in \mathit{interval}(valid(r[i])), \\
&\quad (G(I_1, \ldots, I_m) \wedge valid(u[i]) = V_i(I_1, \ldots, I_m))\}
\end{aligned}
\tag{6.28}
$$

Note that the resulting set may contain many value-equivalent tuples. With this function, we can now define the derivation operator.

$$
\hat{\delta}_{G,\ V_1,\ \ldots,\ V_m}(R) \triangleq \mathbf{NotNull}(\mathbf{Reduce}(\mathbf{Apply}(G, V_1, \ldots, V_m, R))) \tag{6.29}
$$

Had we had disallowed set-valued timestamps, the derivation operator could have been replaced by two simpler operators, analogous to the selection and projection operators, that would have performed tuple selection and attribute projection in terms of the valid components, rather than the value components, of attributes. But disallowing set-valued timestamps would have required that the algebra support value-equivalent tuples, which would have prevented the algebra from having several other, more highly desirable properties.

The snapshot operator $\widehat{SN}$ computes a snapshot relation valid at a specified time τ. If only a subset of attributes is valid at τ, that tuple is not selected. Assume that tuples in R have n attributes.

$$\widehat{SN}_\tau(R) \triangleq \{(value(r[1]),\ \ldots,\ value(r[n])) \mid R(r) \wedge \forall A \in \mathcal{N}(\tau \in valid(r[A]))\} \tag{6.30}$$

EXAMPLE 44 $\mathrm{S}_4 = \widehat{SN}_{\text{2-1-92:12:00PM}}(\mathrm{S}_3) = \{(\text{IBM},\ 23,\ \text{Melanie},\ \text{IBM},\ 20)\}$ □

The dual is the AT operator, which converts a snapshot relation to its valid-time analogue considered valid at the specified time τ.

$$AT_\tau(R') \triangleq \{r \mid \exists r' \in R'\, \forall A \in \mathcal{N}\ (r'[A] = value(r[A]) \wedge valid(r[A]) = \{\tau\})\} \tag{6.31}$$

EXAMPLE 45
$$\begin{aligned}\mathrm{S}_5 &= AT_{\text{2-1-92:12:00PM}}(\mathrm{S}_4)\\ &= \{\langle(\text{IBM}, \{\text{2-1-92:12:00PM}\}),\ (23, \{\text{2-1-92:12:00PM}\}),\\ &\qquad (\text{Melanie}, \{\text{2-1-92:12:00PM}\}),\ (\text{IBM}, \{\text{2-1-92:12:00PM}\}),\\ &\qquad (20, \{\text{2-1-92:12:00PM}\})\rangle\ \}\end{aligned}$$ □

6.4.4 Aggregates

Klug introduced an approach to handle aggregates in the snapshot algebra [Klug 1982]. His approach makes it possible to define aggregates, in particular, non-unique aggregates, in a rigorous fashion. We use his approach to define two aggregate operators for the algebra, $\widehat{A}$, which calculates non-unique aggregates, and $\widehat{AU}$, which calculates unique aggregates. These two valid-time aggregate operators serve as the valid-time counterpart of both scalar aggregates and aggregates with a by clause.

The aggregate operators must contend with a variety of demands that surface as parameters (subscripts) to the operators. First, a specific aggregate (e.g., `count`) must be specified. Secondly, the attribute over which the aggregate is to be applied must be stated and the aggregation window function must be indicated. Finally, to accommodate partitioning, where the aggregate is applied to partitions of a relation, a set of partitioning attributes must be given. These demands complicate the definitions of $\widehat{A}$ and $\widehat{AU}$, but at the same time ensure some degree of generality to these operators.

The aggregate operator is denoted by $\widehat{A}_{f,\ w,\ N,\ X}(Q,\ R)$. R is a valid-time relation of m-tuples over the relation scheme $\mathcal{N}_R$. $N \in \mathcal{N}_R$ is the attribute on which the aggregate is applied. Q supplies the values that partition R. X denotes the attributes on which the partitioning is applied, with the restrictions that $\mathcal{N}_Q \subseteq \mathcal{N}_R$ and $\{N\} \cup X \subseteq \mathcal{N}_Q$. The schema of the result consists of the attributes of R along with an additional attribute, the computed aggregate.

Assume, as does Klug, that for each aggregate operation (e.g., `count`) we have a family of scalar aggregates that performs the indicated aggregation on R (e.g., COUNT_{N_1}, COUNT_{N_2}, …, COUNT_{N_m}, where COUNT_{N_a} counts the (possibly duplicate) values of attribute N_a of R). The particular scalar aggregate is denoted by f. w represents an aggregation window function.

If X is empty, the valid-time aggregate operators simply calculate a single distribution of scalar values over time for an arbitrary aggregate applied to attribute N of relation R. In this case, the tuples in Q are ignored.

EXAMPLE 46 *How many shares of stock does Melanie own?*

$$\hat{\pi}_{Sum}(\widehat{A}_{sum,0,Shares,\emptyset}(\emptyset,\ \hat{\sigma}_{Client=\texttt{"Melanie"}}(Own)))$$

The TQuel version of this query was given in Example 15 in Section 6.2.6. Since this aggregate is an instantaneous aggregate, the aggregate window function is the constant function returning 0. □

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 of an aggregate function in conventional database query languages (e.g., the attributes in the `GROUP BY` clause in SQL [IBM 1981]). Generally $X = \mathcal{N}_Q$ and $Q = \hat{\pi}_X(R)$, but these constraints are not dictated by the formal definition of $\widehat{A}$.

EXAMPLE 47 *Calculate the rate of the Price attribute, partition by the Stock attribute, for each month.*

$$\widehat{A}_{rate,\ month,\ Price,\ \{Stock\}}(\hat{\pi}_{Stock}(Stocks), Stocks)$$

The values to partition the relation are also drawn from *Stocks*. □

6.4.5 Accommodating Transaction Time

Two aspects of supporting transaction time in the algebra must be considered, handling evolution of a database's *content* and handling evolution of a database's *schema*. To handle evolution of the contents of a database containing snapshot, transaction-time, valid-time, and bitemporal relations, we define a relation to be a sequence of snapshot or valid-time states, indexed by transaction time [McKenzie & Snodgrass 1987]. Snapshot and valid-time relations are modeled as single-element sequences. Timeslice operators, to be defined shortly, make past states available in the algebra.

Evolution of a database's schema is associated solely with transaction time. For example, a person's marital status is a (time-varying) aspect of reality, but the decision whether to record marital status, recorded in the schema, is a (time-varying) aspect of the database. We add the relation schemas to the domain of database states [McKenzie & Snodgrass 1990]. Also, as shown in Section 6.2.9, not only the state, but also the *class* of a relation (snapshot, valid-time, etc.), as well as the signature (that is, the attribute names and their associated domains), may change over time. Hence, the representation of a relation manipulated by the algebra must include the current class, signature and state, as well as the signature and state for the intervals of transaction time during which the relation was persistent, i.e., when the class was either transaction-time or bitemporal.

EXAMPLE 48 *The following is the* `Stocks` *relation, with all of its components.*
Class sequence:
(TRANSACTION-TIME, 9-3-92:8:35AM, 10-17-92:4:48PM)
(UNDEFINED, 10-17-92:4:49PM, until changed)

Signature sequence:
((Stock→char, Price→monetary), 6-2-91:1:35PM)
((Stock→char, Price→monetary, NumTraded→integer), 8-19-91:11:31AM)

State sequence:

({⟨(IBM, {6-3-91:11:23AM..forever}), (8, {6-3-91:11:23AM..10-1-91:9:28AM}),
(0, {beginning..forever})⟩,
⟨(IBM, {6-3-91:11:23AM..forever}), (10, {10-1-91:9:30AM..10-2-91:2:14PM}),
(0, {beginning..forever})⟩,
⟨(IBM, {6-3-91:11:23AM..forever}), (12, {10-2-91:2:15PM..10-7-91:10:06AM}),
(0, {beginning..forever})⟩,
⟨(IBM, {6-3-91:11:23AM..forever}), (16, {10-7-91:10:07AM..10-15-91:4:34PM,
10-30-91:4:57PM..11-2-91:12:52PM}),
(0, {beginning..forever})⟩,
⟨(IBM, {6-3-91:11:23AM..forever}), (19, {10-15-91:4:35PM..10-30-91:4:56PM}),
(0, {beginning..forever})⟩,
⟨(IBM, {6-3-91:11:23AM..forever}), (25, {11-2-91:12:53PM..11-5-91:2:01PM}),
(0, {beginning..forever})⟩,
⟨(IBM, {6-3-91:11:23AM..forever}), (27, {11-5-91:2:02PM..12-3-91:8:43AM}),
(0, {beginning..forever})⟩,
⟨(IBM, {6-3-91:11:23AM..forever}), (23, {12-3-91:8:44AM..12-29-91:9:00AM,
1-3-92:4:28PM..forever}),
(0, {beginning..forever})⟩,
⟨(IBM, {6-3-91:11:23AM..forever}), (21, {12-29-91:9:01AM..1-3-92:4:27PM}),
(0, {beginning..forever})⟩}, 9-3-92:8:35AM)
...

The class sequence contains two elements, the signature sequence, two elements, and the state sequence as many elements as there were transactions executed on this relation between September 3, 1992 and October 17, 1992. Before September 3, 1992, the class of the `Stocks` relation was VALID-TIME, and so no old states were retained. □

Two new algebraic operators are available to select a particular snapshot or valid-time state from the sequence recorded in the relation. The snapshot transaction timeslice operator ρ has one argument, a relation name I, and a subscript N designating a transaction number. It retrieves from the relation I the snapshot state current at transaction time N. Note that the timeslice operator does *not* take relations as arguments; it does, however, evaluate to a (snapshot) relation.

EXAMPLE 49 *Retrieve the current state of the* `Stocks` *relation.*

ρ_{now}(`"Stocks"`) □

Similarly, the valid-time transaction timeslice operator $\hat{\rho}_N(I)$ retrieves from the bitemporal relation I the valid-time state current after the transaction at time N.

The algebra is embedded in a language that supports seven commands for database update. BEGIN_TRANSACTION, COMMIT_TRANSACTION and ABORT_TRANSACTION provide both single-command and multiple-command transactions; the latter is treated as an atomic update operation, whether it changes one or several relations. (Like Quel, TQuel treats each statement as a transaction.)

The DEFINE_RELATION command assigns a new class and signature, along with an empty state, to an undefined relation.

EXAMPLE 50 *Define the* `Stocks` *relation.*

```
BEGIN_TRANSACTION
DEFINE_RELATION("Stocks", VALID-TIME, (Stock→char, Price→monetary))
COMMIT_TRANSACTION
```

Example 1 in Section 6.2.1 illustrates the equivalent TQuel ***create*** statement. □

The MODIFY_RELATION command changes the current class, signature, and state of a defined relation. This command supports several TQuel statements. The ***append***, ***delete***, and ***replace*** TQuel statements change the state of a relation. All three can be translated into appropriate MODIFY_RELATION commands.

EXAMPLE 51 *On July 15, 1992 at 3PM, Melanie bought 20 shares of DEC stock.*

```
BEGIN_TRANSACTION
MODIFY_RELATION("Own", *, *,
        [VALID-TIME,(Client→char, Stock→char, Shares→int),
                {⟨("Melanie", {7-15-92:3:00PM..forever}),
                  ("DEC", {7-15-92:3:00PM..forever}),
                  (20, {7-15-92:3:00PM..forever})⟩} ]
        ∪̂ρ̂_now("Stocks"))
COMMIT_TRANSACTION
```

This is equivalent to the TQuel statement in Example 27. A "*" implies that the previous value, in this case the class and schema, should be retained. The "[· · ·]" denotes a constant relation, in this case a single tuple with three attributes; note that a constant relation includes its state and schema. □

The MODIFY_RELATION command may also be used to change the signature.

EXAMPLE 52 *Add a* `NumTraded` *attribute to the* `Stocks` *relation.*

```
BEGIN_TRANSACTION
MODIFY_RELATION("Stocks", *,
        (Stock→char, Price→monetary, NumTraded→integer),
        ρ̂_now("Stocks")
        ×̂[VALID-TIME, (NumTraded→integer), {⟨(0, {beginning..forever})⟩}])
COMMIT_TRANSACTION
```

This is equivalent to Example 32. The third argument to MODIFY_RELATION provides the new signature, and the fourth, a new valid-time state consistent with this signature. □

EXAMPLE 53 *The* `Stocks` *relation should also record all errors.*

> BEGIN_TRANSACTION
> MODIFY_RELATION("`Stocks`", BITEMPORAL, *, $\hat{\rho}_{now}$("`Stocks`"))
> COMMIT_TRANSACTION

This is equivalent to Example 33. □

The DESTROY command is the counterpart of the DEFINE_RELATION command. It either physically or logically deletes from the database the current class, signature, and state of the relation, depending on the relation's class when the command is executed.

EXAMPLE 54 *Remove the* `Stocks` *relation.*

> BEGIN_TRANSACTION
> DESTROY("`Stocks`")
> COMMIT_TRANSACTION

This is equivalent to Example 34. Because `Stocks` is persistent, this command simply appends to the class sequence in the relation the triple

(UNDEFINED, 10-17-92:4:49PM, until changed)

Nothing is physically deleted! □

The RENAME_RELATION command binds the current class, signature, and state of a relation to a new identifier.

We assume that the above commands execute in the context of a single, previously created database. Hence, no commands are necessary to create or delete the database. Since we are considering modeling transaction time from a functional, rather than from a performance, viewpoint, commands affecting access methods, storage mechanisms, or index maintenance are also not relevant.

The full formal semantics of the timeslice operators and the commands introduced above is given elsewhere [McKenzie & Snodgrass 1990]. Allowing a database's schema, as well as its contents, to change increases the complexity of the language. If we allow the database's schema to change, an algebraic expression that is semantically correct for the database's schema when one command executes may not be semantically correct for the database's schema when another command executes. We need a mechanism for identifying semantically incorrect algebraic expressions relative to the database's schema when each command executes and a way of ensuring that the schema and contents of the database state resulting from the command's execution are compatible. To identify semantically incorrect expressions, we introduced a *semantic type system* and augmented the semantics of the commands to do type-checking [McKenzie & Snodgrass 1990]. We chose denotational semantics to define the language because denotational semantics combines a powerful descriptive notation with rigorous mathematical theory [Gordon 1979, Stoy 1977], permitting the precise definition of the database's state.

6.4.6 Valid-time Indeterminacy

As with the tuple calculus, the extensions to the algebra to support histroical indeterminacy are quite minimal. Three basic changes are required, though no new operators or commands are needed. The first is to accommodate indeterminacy spans and probabilities in both the schema and in the tuples themselves, as all combinations are possible. In the schema, this information may be recorded as another component, a sequence indexed by transaction time.

The second change is to add an additional subscript to the timeslice operators, specifying a range credibility between 0 and 100. The modified semantics utilizes the two shrinking functions mentioned in Section 6.3.6.

The third change adds an additional parameter, the ordering plausibility, to temporal predicates and constructors mentioned in the derivation operator.

EXAMPLE 55 *What stocks did Melanie probably own?*

$$\hat{\delta}_{overlap(70,\, Client,\, |\texttt{1AM}|),\, |\texttt{1AM}|}(\hat{\sigma}_{Client=\texttt{"Melanie"}}(\hat{\rho}_{now,\, 100}(\texttt{"Own"}))) \quad \square$$

6.4.7 Properties of the Algebra

An important property of an algebra is that it is *closed*, that is, all operators produce valid objects, in this case valid-time relations.

Theorem 2 *The valid-time algebra is closed.*

A relational algebra is said to be *complete* if it is at least as expressive as the snapshot algebra [Codd 1972].

Theorem 3 *The valid-time algebra is complete.*

We now examine whether the valid-time algebra is in some sense a consistent extension of the snapshot algebra. An algebra is said to *reduce* to the snapshot algebra if taking a snapshot of the result of applying a valid-time operator on one or two valid-time relations is identical to the result of applying the analogous snapshot operator to the snapshots (at the same times) of the valid-time relation(s). Because the temporal algebra allows tuples that contain attributes of differing timestamps, it satisfies this property only through the introduction of distinguished nulls when taking snapshots. We avoid this problem by proving a weaker property: we restrict reducibility to operations on valid-time relations that have identical timestamps for all of their attributes, termed *homogeneous relations* [Gadia 1988].

Theorem 4 *The valid-time operators $\hat{\cup}$, $\hat{-}$, $\hat{\times}$, $\hat{\sigma}$ and $\hat{\pi}$ reduce to their snapshot counterparts when their arguments are homogeneous.*

The language in which the algebra is embedded also has some nice properties (proofs appear elsewhere [McKenzie & Snodgrass 1990]).

Theorem 5 *The language is a natural extension of the relational algebra for database query and update.*

By natural extension, we mean that our semantics subsumes the expressive power of the relational algebra for database query and update. Expressions in the language are a strict superset of those in the relational algebra. Also, if we restrict the class of all relations to UNDEFINED and SNAPSHOT, then a natural extension implies that (a) the signature and state sequences of a defined relation will have exactly one element each: the relation's current signature and state; (b) a new state always will be a function of the current signature and state of defined relations via the relational algebra semantics; and (c) deletion will correspond to physical deletion.

The next property argues that the semantics is minimal, in a specific sense. Other definitions of minimality, such as minimal redundancy or minimal space requirements, are more appropriate for the physical level, where actual data structures are implemented, than for the algebraic level.

Theorem 6 *The semantics of the language minimizes the number of elements in a relation's class, signature, and state sequence needed to record the relation's current class, signature, and state and its history as a transaction-time or bitemporal relation.*

Finally, we ensure that the language accommodates implementations that use write-once-read-many (WORM) optical disk to store non-current class, signature, and state information.

Theorem 7 *Each transaction changes only a relation's class, signature, and state current at the start of the transaction.*

6.4.8 Correspondence with the Calculus

We now show that the valid-time algebra defined above has the expressive power of the TQuel facilities that support valid time.

Theorem 8 *Every TQuel retrieve statement of the form of (6.14) found is equivalent to an expression in the valid-time algebra of the form*

$$R_{k+1} = \hat{\pi}_{N_{i_1},\, a_1,\, \ldots,\, N_{i_n},\, a_n}(\hat{\delta}_{\Gamma_\tau,\, \Phi_v}(\hat{\sigma}_{\psi'}(\hat{\rho}_{\Phi_\alpha}(R_1)\hat{\times}\ldots\hat{\times}\hat{\rho}_{\Phi_\alpha}(R_k)))) \qquad (6.32)$$

EXAMPLE 56 The algebraic equivalent of the TQuel query in Example 15 in Section 6.2.5, listing the current worth of Melanie's stocks, is

$$\hat{\pi}_{Stock, Shares \cdot Price}(\mathrm{S}_3),$$

where S_3 was defined in Example 43 on page 27. The full algebraic expression is

$$\hat{\pi}_{Stock, Shares \cdot Price}(\hat{\delta}_{(Price \cap Shares) \cap now \neq \emptyset,\ Price \cap Shares}(\\ \hat{\sigma}_{Client=\texttt{"Melanie"} \wedge Stock=Stock}(\hat{\rho}_{now}(\texttt{"Stocks"})\hat{\times}\hat{\rho}_{now}(\texttt{"Own"})))). \qquad \square$$

Applying the semantics of aggregation and valid-time indeterminacy yields the following, stronger result.

Theorem 9 *Every TQuel retrieve statement has an equivalent expression in the valid-time algebra.*

EXAMPLE 57 *What stocks have doubled in price over the last month?*

$$\hat{\pi}_{Stock,Price}(\hat{\delta}_{(Stock \cap Rate) \cap now \neq \emptyset,\ Stock \cap Rate}($$
$$\hat{\sigma}_{Rate \geq 2}(\widehat{A}_{rate,\ month,\ Price,\ \{Stock\}}(\hat{\pi}_{Stock}(\hat{\rho}_{now,\ 100}(\texttt{"Stocks"})),$$
$$\hat{\rho}_{now,\ 100}(\texttt{"Stocks"}))))) \qquad \square$$

In a similar fashion, by using the DEFINE_RELATION, MODIFY_RELATION and DESTROY commands, one can construct equivalent algebraic statements for the TQuel create, delete, append, replace, modify, and destroy statements, as are given elsewhere [McKenzie 1988]. This leads to the following central result.

Theorem 10 *The language formed by embedding the valid-time algebra in the commands used to support transaction time has the expressive power of TQuel.*

It turns out that the dual does not hold. For two valid-time relations R_1 and R_2 with at least two tuples that differ in their timestamps, consider the algebraic expression $R_1 \hat{\times} R_2$. Because the semantics of TQuel requires that all attributes within a tuple be associated with identical valid times, this algebraic expression has no counterpart in TQuel, yielding the following result.

Theorem 11 *The temporal algebraic language is strictly more powerful than TQuel.*

Practically speaking, though, this additional power is not needed, as TQuel would be the language of choice for users, with queries translated to the algebra for execution.

6.4.9 Summary

We first introduced *valid-time relations*, in which attribute values are associated with set-valued timestamps. We then defined twelve valid-time operators.

- Five operators are analogous to the five standard snapshot operators: union ($\hat{\cup}$), difference ($\hat{-}$), Cartesian product ($\hat{\times}$), selection ($\hat{\sigma}$), and projection ($\hat{\pi}$).
- The derivation operator ($\hat{\delta}$) effectively performs selection and projection on the valid-time dimension by replacing the timestamp of each attribute of selected tuples with a new timestamp.
- Snapshot ($\widehat{SN}$) and AT convert between snapshot and valid-time relations.
- Aggregation ($\widehat{A}$) and unique aggregation ($\widehat{AU}$) serve to compute a distribution of single values over time for a collection of tuples.

- The snapshot transaction-time timeslice (ρ) and valid-time transaction timeslice ($\hat{\rho}$) operators serve to generalize the algebra to handle bitemporal relations.

We should mention several other operators that can exist harmoniously with these twelve operators. Intersection ($\hat{\cap}$) and Θ-join ($\hat{\bowtie}_{\Theta}$) can be defined in terms of the five basic operators, in an identical fashion to the definition of their snapshot counterparts. Valid-time natural join ($\hat{\bowtie}$) and quotient ($\hat{\div}$) can't be defined in this way, because both involve projection, an operation whose semantics in the valid-time algebra is substantially different from its semantics in the snapshot algebra. Small, but important, changes must be made to the definitions to handle properly the temporal dimension [McKenzie & Snodgrass 1991A]. It is also possible to extend the algebra in a consistent fashion to support periodicity [Lorentzos & Johnson 1988], multi-dimensional valid timestamps [Bhargava & Gadia 1989, Bhargava & Gadia 1991, Gadia & Yeung 1988], and non-first-normal-form valid-time relations with an arbitrary level of nesting [Roth et al. 1988, Schek & Scholl 1986, Tansel & Garnett 1989, Özsoyoğlu et al. 1987].

For valid-time indeterminacy, ordering plausibility is supported by an additional argument in temporal predicates and constructors within the derivation operator. A second subscript on the timeslice operators supports range credibility.

We also discussed seven commands that embed the algebra and permit evolution of the contents of the database as well as its schema: DEFINE_RELATION, MODIFY_RELATION, DESTROY, RENAME_RELATION, BEGIN_TRANSACTION, COMMIT_TRANSACTION and ABORT_TRANSACTION.

Finally, we listed several important properties of the algebra, and showed that its expressive power is greater than that of TQuel, allowing it to serve as the operational counterpart of this declarative query language.

6.5 Implementation

A temporal algebra is a critical part of a DBMS that supports time-varying information. Such an algebra can serve as (1) an appropriate target for a temporal query language processor, (2) an appropriate structure on which to perform optimization, and (3) an appropriate executable formalism for the DBMS to interpret to execute queries. The previous section showed that the valid-time algebra has the expressive power of TQuel, thus satisfying the first objective just listed. In this section we discuss the other two objectives, focusing on query optimization and page structure. Elsewhere we also examine incremental update of materialized views [McKenzie 1988].

In particular, we show that all but one of the traditional tautologies used in query optimization hold for the algebra. Various implementation aspects are also considered. We show how the algebra may utilize a page layout that is quite similiar to that used by conventional DBMS's.

6.5.1 Query Optimization

Query optimization concerns the problem of selecting an efficient query plan for a query from the set of all its possible query plans. This problem for snapshot queries has been studied extensively and heuristic algorithms have been proposed for selection of a near optimal query plan based on a statistical description of the database and a cost model for query plan execution [Hall 1976, Jarke & Koch 1984, Krishnamurthy et al. 1986, Selinger et al. 1979, Smith & Chang 1975, Stonebraker et al. 1976, Wong & Youssefi 1976, Yao 1979].

One important aspect of local query optimization is the transformation of one query plan into an equivalent, but more efficient, query plan. The size of the search space of equivalent query plans for a snapshot query is determined in part by the algebraic equivalences available in the snapshot algebra. Both Ullman and Maier identify equivalences based on those in set theory [Enderton 1977] that are available in the snapshot algebra for query plan transformation and describe their usefulness to query optimization [Maier 1983, Ullman 1988]. We now examine which of these equivalences hold.

For the theorems that follow, assume that Q, R, and S are valid-time relations.

Theorem 12 *The following equivalences hold for the valid-time algebra.*

$$\begin{array}{rlrl}
Q \hat{\cup} R & \equiv R \hat{\cup} Q & Q \hat{\times} R & \equiv R \hat{\times} Q \\
\hat{\sigma}_{F_1}(\hat{\sigma}_{F_2}(Q)) & \equiv \hat{\sigma}_{F_2}(\hat{\sigma}_{F_1}(Q)) & Q \hat{\cup} (R \hat{\cup} S) & \equiv (Q \hat{\cup} R) \hat{\cup} S \\
Q \hat{\times} (R \hat{\times} S) & \equiv (Q \hat{\times} R) \hat{\times} S & Q \hat{\times} (R \hat{\cup} S) & \equiv (Q \hat{\times} R) \hat{\cup} (Q \hat{\times} S) \\
\hat{\sigma}_F(Q \hat{\cup} R) & \equiv \hat{\sigma}_F(Q) \hat{\cup} \hat{\sigma}_F(R) & \hat{\sigma}_F(Q \hat{-} R) & \equiv \hat{\sigma}_F(Q) \hat{-} \hat{\sigma}_F(R) \\
\hat{\pi}_X(Q \hat{\cup} R) & \equiv \hat{\pi}_X(Q) \hat{\cup} \hat{\pi}_X(R) & &
\end{array}$$

Theorem 13 *The distributive property of Cartesian product over difference, or* $Q \hat{\times} (R \hat{-} S) \equiv (Q \hat{\times} R) \hat{-} (Q \hat{\times} S)$*, does* not *hold for the valid-time algebra.*

Ullman identifies several conditional equivalences involving selection and projection that can be used in optimizing snapshot queries [Ullman 1988]. These conditional equivalences also hold in the valid-time algebra. Elsewhere we give eight additional equivalences involving the derivation operator that have no snapshot counterparts [McKenzie & Snodgrass 1991A]. No equivalences are available that involve the derivation operator together with union, difference, or projection: the derivation operator doesn't commute with projection or distribute over union or difference, even conditionally, as these operators may change attribute timestamps.

In summary, all the above non-conditional and conditional equivalences may be used, along with statistical descriptions of valid-time databases and cost models for query plan execution, to optimize individual temporal queries. Because all but one of the equivalences that hold for the snapshot algebra also hold for the valid-time algebra, the search space of equivalent query plans for a temporal query should be comparable in size to that for an analogous snapshot query. Hence, the valid-time algebra does not limit the practical use of query plan transformation as an optimization technique for temporal queries. Also, most algorithms for optimization of snapshot queries may be extended to optimize temporal queries by taking into account the possible presence of derivation operators in query plans.

6.5.2 Page Structure

A valid-time tuple is more complex than a conventional tuple, because timestamps are sets. As first normal form (1NF) dictates that each value of a tuple be atomic [Elmasri & Navathe 1989], valid-time relations cannot be considered to be in 1NF. However, they are close, in that the value component of an attribute *is* atomic. One simple means of retaining much of the simplicity of conventional relations is to implement the set of chronons forming the timestamp of an attribute as a linked list of intervals, each represented with an *interval cell* containing a starting timestamp, an ending timestamp, and a pointer to the next interval. An attribute's timestamp then becomes a fixed-length pointer field. For page sizes under 4K bytes, a single byte sufficies for a pointer; if overflow pages are permitted then two bytes are required for the pointer. Using interval lists, fixed-length tuples remain of fixed length even when timestamps are added, and conventional techniques, e.g., of attribute-value space compression and null value representation, still apply. Efficient implementations for determinate as well as indeterminate timestamps exist [Dyreson & Snodgrass 1992B].

Various space management approaches are available to contend with the interval lists now present [Hsu 1992]. If tuples are fixed-length, then the page may be partitioned into fixed-length slots, each to be occupied either by a tuple or by several interval cells. Variable-length tuples are often handled by placing the tuples at the top of the page growing down and tuple headers at the bottom of the page growing up, with free space in the middle [Stonebraker et al. 1976]. The interval lists also vary in size. They can either be allocated in the same space as the tuples, or the tuple headers can be pre-allocated (since they are short, 1 to 2 bytes, preallocation will not waste much space), and the intervals can start at the bottom of the page and grow up. In all cases, compaction will be necessary upon deallocation of an interval [Knuth 1973]. Interval cells can be clustered into *blocks* to reduce the overhead of the next block pointer. For 16-byte intervals, 2 to 5 cells per block are indicated for several linked-list length distributions [Hsu & Snodgrass 1991].

The timestamps for time-invariant attributes may be either stored as a special value, distinguishable from an interval pointer, that represents the set containing all chronons, or not stored at all, but instead indicated as time-invariant in the schema. Several attributes often share the same timestamp; again, this can be indicated in the schema, with only one interval pointer allocated for the group (this implementation shares some aspects with the multi-homogeneous data model [Gadia 1986]), or can be represented at the extension level by having multiple interval pointers pointing to the same interval list head cell (though care must be taken when modifying such shared interval lists).

If the algebra is used to implement TQuel, then a conversion will be necessary between tuple timestamping, where each tuple is associated with a single interval, and attribute-value timestamping, in which each attribute is associated with potentially multiple intervals. This conversion is formalized in a transformation function **T** [McKenzie & Snodgrass 1991A]; it is similar to the *Pack* operation (also termed *nest*) proposed for non-1NF relations [Tansel 1986, Özsoyoğlu et al. 1987].

There are a variety of ways to effect this transformation. The brute-force method is to first cluster the relation on a key, perhaps by sorting the relation, so that all of the versions are collected on the same page, then link up the intervals, distributing them to the attributes. Since redundant attribute values occur in a tuple timestamped representation, the space requirements will decrease during this conversion, guaranteeing that no new overflow pages will result. If we record in the schema that all attributes contain the same timestamp, then we need not duplicate interval lists for each attribute. The conversion can even be done in parallel with any of the temporal operators. When the operator fetches another tuple, the interval list can be constructed and passed to the operator, assuming that the underlying relation was clustered on the key.

Once an algebraic expression has computed a result relation, it must be converted back into a tuple timestamped representation. This step is even easier than the other direction. The TQuel semantics presented in Section 6.4.8 ensures that the timestamps of all of the attributes are identical within a tuple, so all that is necessary is to make a duplicate of each tuple for each interval in the interval list. This expansion also can be done within any of the temporal operators. The conversion is similar to the *Unpack* operation (also termed *unnest*) in non-1NF relations. It has been shown that applying the Pack operation followed by Unpack operation, i.e., performing the empty algebraic expression on a tuple-timestamped relation, produces the original relation [Jaeschke & Schek 1982].

Finally, there is no reason why a relation *logically* timestamped on a tuple basis with single intervals can't be stored *physically* as timestamped with a set (linked list) of intervals, in concert with the space optimization of utilizing only one interval pointer for the entire tuple. This storage structure requires conversion only on display, which is much less time-critical than conversion on access and on storage.

6.6 Summary

This chapter has presented the syntax and formal semantics for the temporal query language TQuel. The discussion proceeded in an incremental fashion for both the syntax and semantics. First, the Quel syntax was presented informally. Temporal analogues for the where clause and the target list were examined. Aggregates, valid-time indeterminacy, and database update were also considered.

After a short review of tuple calculus, the semantics of temporal constructors was described as functions on time values or pairs of time values, ultimately yielding a time value. A transformation system provided the semantics of temporal predicates, yielding a conventional predicate on the participating tuples. The semantics of the retrieve statement without aggregates was presented. This semantics was extended to accommodate aggregation, valid-time indeterminacy, and update. The semantics reduces to the standard Quel semantics.

We then presented a temporal relational algebra. The design of a relational algebra incorporating the time dimension that simultaneously satisfies many desirable properties is a surprisingly difficult task. Since all desirable properties of temporal algebras are not compatible [McKenzie & Snodgrass 1991B], the best that can be hoped for is not an algebra with all possible desirable properties but an algebra with a maximal subset of the most desirable properties.

We defined our algebra as a straightforward extension of the conventional relational algebra. The algebra includes operators that are analogous to the five standard snapshot operators, a derivation operator, operators to perform aggregation and unique aggregation, operators to convert between snapshot and valid-time relations, and two timeslice operators. Minor extensions to the derivation and timeslice operators accommodate valid-time indeterminacy. The algebraic language contains seven commands to effect evolution of the contents of the database as well as its schema.

The algebra was shown to be closed, complete, minimal, and snapshot reducible. It was also shown to have the expressive power of TQuel. As such, the algebra provides an executable equivalent of a declarative query language. The algebra satisfies all but one of the commutative, associative, and distributive tautologies involving union, difference, and Cartesian product as well as the non-conditional commutative laws involving selection and projection. Additional equivalences involving the derivation operator also hold. Hence, most existing optimization algorithms may be naturally extended to optimize temporal queries. Conversion between valid-time relations and the tuple-timestamping assumed by TQuel is simple and efficient. Finally, we discussed representations of valid-time relations on secondary storage that are straightforward extensions of those of conventional relations.

6.7 TQuel Syntax

In this syntax specification, we use an extended BNF in which "$\{\cdots \mid \cdots\}$" denotes one of the listed alternatives, "$\{\cdots\ \}^?$" denotes optional syntax, and "$\{\ \cdots$ ',' $\}^+$" denotes a list of one or more items, separated with commas.

⟨statement⟩ ::= ⟨create stmt⟩ | ⟨range stmt⟩ | ⟨retrieve stmt⟩
| ⟨append stmt⟩ | ⟨delete stmt⟩ | ⟨replace stmt⟩
| ⟨modify stmt⟩ | ⟨index stmt⟩ | ⟨set stmt⟩
| ⟨destroy stmt⟩

⟨create stmt⟩ ::= `create` $\{$ `persistent` $\}^?$ ⟨history⟩ ⟨relation name⟩
'(' $\{$ ⟨column name⟩ ⟨is⟩ ⟨type⟩ ',' $\}^+$ ')'

⟨history⟩ ::= ϵ | $\{$ `indeterminate` $\}^?$ $\{$ `event` | `interval` $\}$

⟨is⟩ ::= ':' | `is`

⟨type⟩ ::= *CHAR* '(' ⟨integer constant⟩ ')' | *I2* | *I4* | *F4* | *F8*
| *interval* | *event* | *span*

⟨integer constant⟩ ::= $\{$ ⟨digit⟩ $\}^{+}$

⟨range stmt⟩ ::= *range of* ⟨tuple variable⟩ *is* ⟨relation name⟩
$\{$ *with credibility* ⟨two digit⟩ $\}^{?}$

⟨two digit⟩ ::= 100 | ⟨digit⟩ | ⟨digit⟩ ⟨digit⟩

⟨digit⟩ ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'
| '8' | '9'

⟨retrieve stmt⟩ ::= *retrieve* ⟨into⟩ '(' ⟨target list⟩ ')' ⟨with clause⟩
⟨valid clause⟩ ⟨retrieve tail⟩

⟨target list⟩ ::= ⟨tuple variable⟩ '.' *all*
| $\{$ ⟨column name⟩ $\{$ '=' ⟨expression⟩ $\}^{?}$
$\{$*as* ⟨calendar name⟩ $\}^{?}$ $\}^{+}$

⟨with clause⟩ ::= ϵ | *with plausibility* ⟨two digit⟩

⟨valid clause⟩ ::= ⟨valid⟩ *during* ⟨i-expression⟩ ⟨with clause⟩
| ⟨valid⟩ *at* ⟨e-expression⟩ ⟨with clause⟩

⟨valid⟩ ::= ϵ | *valid*

⟨retrieve tail⟩ ::= ⟨where clause⟩ ⟨when clause⟩ ⟨as of clause⟩

⟨into⟩ ::= ϵ | *unique* | ⟨relation⟩ | *into* ⟨relation⟩
| *to* ⟨relation⟩

⟨where clause⟩ ::= ϵ | *where* ⟨bool expression⟩

⟨when clause⟩ ::= ϵ | *when* ⟨temporal pred⟩ ⟨with clause⟩

⟨as-of clause⟩ ::= ϵ | *as of* ⟨e-expression⟩ ⟨through clause⟩

⟨through clause⟩ ::= ϵ | *through* ⟨e-expression⟩

⟨append stmt⟩ ::= *append* ⟨to⟩ ⟨target list⟩ ⟨mod stmt tail⟩

⟨to⟩ ::= ⟨relation⟩ | *to* ⟨relation⟩

⟨delete stmt⟩ ::= *delete* ⟨tuple variable⟩ ⟨mod stmt tail⟩

⟨replace stmt⟩ ::= *replace* ⟨tuple variable⟩ ⟨target list⟩ ⟨mod stmt tail⟩

⟨mod stmt tail⟩ ::= ⟨valid clause⟩ ⟨where clause⟩ ⟨when clause⟩

```
⟨e-expression⟩          ::= ⟨event element⟩
                          | begin of ⟨i-expression⟩
                          | end of ⟨i-expression⟩
                          | '(' ⟨e-expression⟩ ')'
⟨i-expression⟩          ::= ⟨interval element⟩
                          | interval '(' ⟨e-expression⟩ ','
                                ⟨e-expression⟩ ')'
                          | ⟨either-expression⟩ overlap ⟨plaus suffix⟩
                                ⟨either-expression⟩
                          | ⟨either-expression⟩ extend ⟨plaus suffix⟩
                                ⟨either-expression⟩
                          | '(' ⟨i-expression⟩ ')'
⟨plaus suffix⟩          ::= ε | '(' ⟨two digit⟩ ')'
⟨either-expression⟩     ::= ⟨e-expression⟩ | ⟨i-expression⟩
⟨event element⟩         ::= ⟨tuple variable⟩ ⟨credibility suffix⟩
                          | '|' ⟨event value⟩ '|' { as ⟨calendar name⟩ }?
                          | present
                          | ⟨event agg⟩ '(' ⟨either-expression⟩
                                ⟨aggregate tail⟩ ⟨with clause⟩ ')'
⟨credibility suffix⟩    ::= ε | '(' ⟨two digit⟩ ')'
⟨event agg⟩             ::= earliest | latest
⟨interval element⟩      ::= ⟨tuple variable⟩ ⟨credibility suffix⟩
                          | '[' ⟨interval value⟩ ']' { as ⟨calendar name⟩ }?
                          | ⟨interval agg⟩ '(' ⟨either-expression⟩
                                ⟨aggregate tail⟩ ⟨with clause⟩ ')'
⟨interval agg⟩          ::= earliest | latest | rising
⟨temporal pred⟩         ::= ⟨either-expression⟩ precede ⟨plaus suffix⟩
                                ⟨either-expression⟩
                          | ⟨either-expression⟩ overlap ⟨plaus suffix⟩
                                ⟨either-expression⟩
                          | ⟨either-expression⟩ equal ⟨plaus suffix⟩
                                ⟨either-expression⟩
                          | ⟨temporal pred⟩ and ⟨temporal pred⟩
                          | ⟨temporal pred⟩ or ⟨temporal pred⟩
```

```
                         | ‘(’ ⟨temporal pred⟩ ‘)’
                         | not ⟨temporal pred⟩

⟨expression⟩            ::= ⟨arithmetic expression⟩ | ⟨user time expression⟩
⟨arithmetic expression⟩ ::= ⟨aggregate term⟩
⟨aggregate term⟩        ::= ⟨aggregate op⟩ ‘(’ ⟨expression⟩ ⟨aggregate tail⟩
                                ⟨with clause⟩ ‘)’
                          | var ‘(’ ⟨e-expression⟩ ⟨aggregate tail⟩
                                ⟨with clause⟩ ‘)’
                          | rate ‘(’ ⟨e-expression⟩ ⟨aggregate tail⟩
                                ⟨per clause⟩ ⟨with clause⟩ ‘)’
⟨aggregate tail⟩        ::= ⟨by clause⟩ ⟨for clause⟩ ⟨retrieve tail⟩
⟨by clause⟩             ::= ε | by { ⟨expression⟩ ‘,’ }+
⟨aggregate op⟩          ::= count | countU | sum | sumU | avg | avgU
                          | stdev | stdevU | any | min | max
                          | first | last
⟨for clause⟩            ::= ε | for each instant | for ever
                          | for each ⟨span element⟩
⟨span element⟩          ::= ‘%’ ⟨span value⟩ ‘%’ { as ⟨calendar name⟩ }?
⟨per clause⟩            ::= ε | per ⟨span element⟩

⟨index stmt⟩            ::= index on ⟨relation name⟩ is ⟨index name⟩
                                ‘(’ { ⟨column⟩ ‘,’ }+ ‘)’
                                { as ⟨index type⟩ }?
⟨index type⟩            ::= snapshot | valid-time | transaction-time
                          | bitemporal

⟨modify stmt⟩           ::= modify ⟨relation name⟩ ⟨modify tail⟩
⟨modify tail⟩           ::= ‘(’ { ⟨column name⟩ { ⟨is⟩ ⟨type⟩ }?
                                          ‘=’ ⟨expression⟩ ‘,’ }+ ‘)’
                                ⟨valid clause⟩ ⟨with clause⟩ ⟨retrieve tail⟩
                          | to { {not}? persistent }?
                                { not valid-time | ⟨history⟩ }?
                          | to { validfrom | validto }?
```

{ ⟨relation name⟩ | *arbitrary* } *distribution*
| *to* { *validfrom* | *validto* $\}^{?}$ *determinate*
| *to* { *validfrom* | *validto* $\}^{?}$ *indeterminate*
span = ⟨span element⟩
| *to* { *hash* | *isam* | *index* }
on { ⟨column name⟩ ‘,’ $\}^{+}$
as { { *not* $\}^{?}$ *persistent* $\}^{?}$
{ { *not* $\}^{?}$ *historical* $\}^{?}$
{ *where fillfactor* ‘=’ ⟨integer constant⟩ $\}^{?}$
| *to accessionlist on* { ⟨column name⟩ ‘,’ $\}^{+}$
where time ‘=’
‘(’ { *all* | { ⟨time⟩ ‘,’ $\}^{+}$ } ‘)’
| *to* { *cellular* | *cluster* | *stack* } *on*
{ ⟨column name⟩ ‘,’ $\}^{+}$
where cellsize ‘=’ ⟨integer constant⟩

⟨time⟩ ::= *validfrom* | *validto* | *transactionfrom*
| *transactionto*

⟨set stmt⟩ ::= *set calendric system* ⟨calendar name⟩
| *set default* ⟨indeterminacy⟩ ‘=’ ⟨two digit⟩

⟨indeterminacy⟩ ::= *range credibility* | *ordering plausibility*

⟨destroy stmt⟩ ::= *destroy* { ⟨relation name⟩ ‘,’ $\}^{+}$

Acknowledgements

The author thanks Ilsoo Ahn, Curtis E. Dyreson, Christian S. Jensen, Edwin L. McKenzie, Jr., Michael Soo, and Juan Valiente for their contributions towards the design of this language, as well as for their assistance in preparing this chapter. Keun Ryu assisted with the implementation of the prototype. Christian S. Jensen's comments were especially detailed and helpful. The author was supported in part by an IBM Faculty Development Award. This research was also supported in part by NSF grants DCR-8402339 and IRI-8902707, by ONR contract N00014-86-K-0680, by a Junior Faculty Development Award from the UNC-CH Foundation, and by the NCR Corporation.

Bibliography

[Ahn 1986] Ahn, I. *Performance Modeling and Access Methods for Temporal Database Management Systems*. PhD. Diss. Computer Science Department, University of North Carolina at Chapel Hill, July 1986.

[Ahn & Snodgrass 1986] Ahn, I. and R. Snodgrass. *Performance Evaluation of a Temporal Database Management System*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Ed. C. Zaniolo. Association for Computing Machinery. Washington, DC: May 1986, pp. 96–107.

[Ahn & Snodgrass 1988] Ahn, I. and R. Snodgrass. *Partitioned Storage for Temporal Databases*. *Information Systems*, 13, No. 4 (1988), pp. 369–391.

[Ahn & Snodgrass 1989] Ahn, I. and R. Snodgrass. *Performance Analysis of Temporal Queries*. *Information Sciences*, 49 (1989), pp. 103–146.

[Anderson 1982] Anderson, T.L. *Modeling Time at the Conceptual Level*, in *Proceedings of the International Conference on Databases: Improving Usability and Responsiveness*. Ed. P. Scheuermann. Jerusalem, Israel: Academic Press, June 1982, pp. 273–297.

[Bhargava & Gadia 1989] Bhargava, G. and S.K. Gadia. *A 2-dimensional temporal relational database model for querying errors and updates, and for achieving zero information-loss*. Technical Report TR#89-24. Department of Computer Science, Iowa State University. Dec. 1989.

[Bhargava & Gadia 1991] Bhargava, G. and S.K. Gadia. *Relational database systems with zero information-loss*. *IEEE Transactions on Knowledge and Data Engineering*, (to appear) (1991).

[Bontempo 1983] Bontempo, C. J. *Feature Analysis of Query-By-Example*, in Relational Database Systems. New York: Springer-Verlag, 1983. pp. 409–433.

[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.

[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 1970] Codd, E.F. *A Relational Model of Data for Large Shared Data Banks*. *Communications of the Association of Computing Machinery*, 13, No. 6, June 1970, pp. 377–387.

[Dyreson & Snodgrass 1992A] Dyreson, C. E. and R. T. Snodgrass. *Historical Indeterminacy*. Technical Report TR 91-30a. Computer Science Department, University of Arizona. Revised Feb. 1992.

[Dyreson & Snodgrass 1992B] Dyreson, C. E. and R. T. Snodgrass. *Time-stamp Semantics and Representation*. TempIS Technical Report 33. Computer Science Department, University of Arizona. Revised May 1992.

[Elmasri & Navathe 1989] Elmasri, R. and S.B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings Pub. Co., 1989.

[Enderton 1977] Enderton, H.B. *Elements of Set Theory*. New York, N.Y.: Academic Press, Inc., 1977.

[Gadia 1986] Gadia, S.K. *Toward a Multihomogeneous Model for a Temporal Database*, in *Proceedings of the International Conference on Data Engineering*. IEEE Computer Society. Los Angeles, CA: IEEE Computer Society Press, Feb. 1986, pp. 390–397.

[Gadia 1988] Gadia, S.K. *A Homogeneous Relational Model and Query Languages for Temporal Databases*. *ACM Transactions on Database Systems*, 13, No. 4, Dec. 1988, pp. 418–448.

[Gadia & Yeung 1988] Gadia, S.K. and C.S. Yeung. *A Generalized Model for a Relational Temporal Database*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery. Chicago, IL: June 1988, pp. 251–259.

[Gordon 1979] Gordon, M.J.C. *The Denotational Description of Programming Languages*. New York-Heidelberg-Berlin: Springer-Verlag, 1979.

[Hall 1976] Hall, P.A.V. *Optimization of Single Expressions in a Relational Data Base System*. *IBM Journal of Research and Development*, 20, No. 3, May 1976, pp. 244–257.

[Held et al. 1975] Held, G.D., M. Stonebraker and E. Wong. *INGRES–A Relational Data Base Management System*, in *Proceedings of the AFIPS National Computer*

Conference. Anaheim, CA: AFIPS Press, May 1975, pp. 409–416.

[Hsu & Snodgrass 1991] Hsu, S.H. and R.T. Snodgrass. *Optimal Block Size for Repeating Attributes*. TempIS Technical Report No. 28. Department of Computer Science, University of Arizona. Dec. 1991.

[Hsu 1992] Hsu, S.H. *Page and Tuple Level Storage Structures for Historical Databases*. TempIS Technical Report No. 34. Computer Science Department, University of Arizona. May 1992.

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

[Jaeschke & Schek 1982] Jaeschke, G. and H.J. Schek. *Remarks on the Algebra of Non First Normal Form Relations*, in *Proceedings of the ACM Symposium on Principles of Database Systems*. 1982.

[Jarke & Koch 1984] Jarke, M. and J. Koch. *Query Optimization in Database Systems*. *ACM Computing Surveys*, 16, No. 2, June 1984, pp. 111–152.

[Jensen & Snodgrass 1992A] Jensen, C. S. and R. Snodgrass. *Temporal Specialization and Generalization, to appear*. *IEEE Transactions on Knowledge and Data Engineering*, (1992).

[Jensen & Snodgrass 1992B] Jensen, C.S. and R. Snodgrass. *Temporal Specialization*, in *Proceedings of the International Conference on Data Engineering*. Ed. F. Golshani. IEEE. Tempe, AZ: Feb. 1992, pp. 594–603.

[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.

[Knuth 1973] Knuth, D.E. *Fundamental Algorithms*. Vol. 1, Second Edition of The Art of Computer Programming. Addison-Wesley, 1973.

[Krishnamurthy et al. 1986] Krishnamurthy, R., H. Boral and C. Zaniolo. *Optimization of Nonrecursive Queries*, in *Proceedings of the Conference on Very Large Databases*. Ed. Y. Kambayashi. Kyoto, Japan: Aug. 1986, pp. 128–137.

[Lorentzos & Johnson 1988] Lorentzos, N. and R. Johnson. *Extending Relational Algebra to Manipulate Temporal Data*. *Information Systems*, 13, No. 3 (1988), pp. 289–296.

[Maier 1983] Maier, D. *The Theory of Relational Databases*. Rockville, MD: Computer Science Press, 1983.

[McKenzie & Snodgrass 1987] McKenzie, E. and R. Snodgrass. *Extending the Relational Algebra to Support Transaction Time*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Ed. U. Dayal and I. Traiger. Association for Computing Machinery. San Francisco, CA: May 1987, pp. 467–478.

[McKenzie 1988] McKenzie, E. *An Algebraic Language for Query and Update of Temporal Databases*. PhD. Diss. Computer Science Department, University of North Carolina at Chapel Hill, Sep. 1988.

[McKenzie & Snodgrass 1990] McKenzie, E. and R. Snodgrass. *Schema Evolution and the Relational Algebra*. *Information Systems*, 15, No. 2, June 1990, pp. 207–232.

[McKenzie & Snodgrass 1991A] McKenzie, E. and R. Snodgrass. *Supporting Valid Time in an Historical Relational Algebra: Proofs and Extensions*. Technical Report TR–91–15. Department of Computer Science, University of Arizona. Aug. 1991.

[McKenzie & Snodgrass 1991B] McKenzie, E. and R. Snodgrass. *An Evaluation of Relational Algebras Incorporating the Time Dimension in Databases*. *ACM Computing Surveys*, 23, No. 4, Dec. 1991, pp. 501–543.

[Navathe & Ahmed 1989] Navathe, S. B. and R. Ahmed. *A Temporal Relational Model and a Query Language*. *Information Sciences*, 49 (1989), pp. 147–175.

[Overmyer & Stonebraker 1982] Overmyer, R. and M. Stonebraker. *Implementation of a Time Expert in a Database System*. *ACM SIGMOD Record*, 12, No. 3, Apr. 1982, pp. 51–59.

[Özsoyoğlu et al. 1987] Özsoyoğlu, G., Z. Özsoyoğlu and V. Matos. *Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions*. *ACM Transactions on Database Systems*, 12, No. 4, Dec. 1987, pp. 566–592.

[Roth et al. 1988] Roth, Mark A., Henry F. Korth and Abraham Silberschatz. *Ex-*

tended Algebra and Calculus for Nested Relational Databases. *ACM Transactions on Database Systems*, 13, No. 4, Dec. 1988, pp. 389–417.

[Schek & Scholl 1986] Schek, H.-J., Scholl, M.H. *The Relational Model with Relation-valued Attributes*. *Information Systems*, 11, No. 2 (1986), pp. 137–147.

[Segev & Shoshani 1987] Segev, A. and A. Shoshani. *Logical Modeling of Temporal Data*, in *Proceedings of the ACM SIGMOD Annual Conference on Management of Data*. Ed. U. Dayal and I. Traiger. Association for Computing Machinery. San Francisco, CA: ACM Press, May 1987, pp. 454–466.

[Selinger et al. 1979] Selinger, P.G., M.M. Astrahan, D.D. Chamberlin, R.A. Lorie and T.G. Price. *Access Path Selection in a Relational Database Management System*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Ed. P.A. Bernstein. Association for Computing Machinery. Boston, MA: 1979, pp. 23–34.

[Smith & Chang 1975] Smith, J.M. and P.Y-T. Chang. *Optimizing the Performance of a Relational Algebra Database Interface*. *Communications of the Association of Computing Machinery*, 18, No. 10, Oct. 1975, pp. 568–579.

[Snodgrass & Ahn 1985] Snodgrass, R. and I. Ahn. *A Taxonomy of Time in Databases*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Ed. S. Navathe. Association for Computing Machinery. Austin, TX: May 1985, pp. 236–246.

[Snodgrass & Ahn 1986] Snodgrass, R. and I. Ahn. *Temporal Databases*. *IEEE Computer*, 19, No. 9, Sep. 1986, pp. 35–42.

[Snodgrass, et al. 1992] Snodgrass, R., S. Gomez and E. McKenzie. *Aggregates in the Temporal Query Language TQuel, to appear*. *IEEE Transactions on Knowledge and Data Engineering*, (1992).

[Snodgrass 1987] Snodgrass, R. T. *The Temporal Query Language TQuel*. *ACM Transactions on Database Systems*, 12, No. 2, June 1987, pp. 247–298.

[Soo & Snodgrass 1992A] Soo, M. and R. Snodgrass. *Multiple Calendar Support for Conventional Database Management Systems*. Technical Report 92-7. Computer Science Department, University of Arizona. Feb. 1992.

[Soo & Snodgrass 1992B] Soo, M. and R. Snodgrass. *Mixed Calendar Query Language*

Support for Temporal Constants. TempIS Technical Report 29. Computer Science Department, University of Arizona. Revised May 1992.

[Soo et al. 1992] Soo, M., R. Snodgrass, C. Dyreson, C. S. Jensen and N. Kline. *Architectural Extensions to Support Multiple Calendars*. TempIS Technical Report 32. Computer Science Department, University of Arizona. Revised May 1992.

[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.

[Stoy 1977] Stoy, Joseph E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Series in Computer Science. The MIT Press, 1977.

[Tandem 1983] Tandem Computers, Inc. *ENFORM Reference Manual*. Cupertino, CA, 1983.

[Tansel & Garnett 1989] Tansel, A. and L. Garnett. *Nested Historical Relations*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. May 1989, pp. 284–293.

[Tansel 1986] Tansel, A.U. *Adding Time Dimension to Relational Model and Extending Relational Algebra*. *Information Systems*, 11, No. 4 (1986), pp. 343–355.

[Ullman 1988] Ullman, J.D. *Principles of Database and Knowledge-Base Systems*. Potomac, Maryland: Computer Science Press, 1988. Vol. 1.

[Wong & Youssefi 1976] Wong, E. and K. Youssefi. *Decomposition - A Strategy for Query Processing*. *ACM Transactions on Database Systems*, 1, No. 3, Sep. 1976, pp. 223–241.

[Yao 1979] Yao, S.B. *Optimization of Query Evaluation Algorithms*. *ACM Transactions on Database Systems*, 4, No. 2, June 1979, pp. 133–155.