15

Valid-Time Selection and Projection

Suchen Hsu, Christian S. Jensen, and Richard T. Snodgrass

1 Introduction

Temporal database management systems should offer user-friendly and powerful means of retrieval of data according to temporal criteria. Selection of tuples according to their *valid times*, which is the times when the information represented by the tuples is valid in the modeled reality [16], is termed *valid-time selection*.

The attributes to be computed and returned from a query are specified in the target list. *Valid-time projection* is the temporal analog of regular projection in, e.g., Quel, and allows for the specification of the implicit timestamps for tuples of the resulting relation.

Valid-time projection mechanisms may be partitioned into two categories. First, valid-time projection may be built implicitly into the operators and constructs of the query language. This leaves the user no freedom in specifying the timestamps of result tuples. Second, valid-time projection may be specified explicitly, as part of an existing clause or in a new clause. This approach allows the user to control the timestamps of result tuples.

The chapter is structured as follows. Initially, we survey previous temporal query language proposals. We then describe six general criteria for good language design. The criteria provide guidelines. However, they may conflict at times, and a single general criterion may be applied in several ways in a concrete design. Then, TSQL2's method for valid-time selection is introduced. Mechanisms that involve the referencing and construction/extraction of various kinds of timestamps are covered first. Then, comparison operators are covered. Finally, miscellaneous aspects of the design are discussed. The general design is first presented, and then central design issues are discussed in more detail.

We then turn to temporal projection. The motivation for adding a separate clause for valid-time projection is discussed, and the choice of defaults is considered in some detail. The chapter ends with a summary.

2 Survey

In the past decade, numerous temporal extensions to the relational data model have been proposed. Most of these data model proposals have included user-level query languages. Most of these languages are not entirely new, but are instead extensions of existing, non-temporal query languages. To illustrate, extensions of SQL [4] 1976] include TOSQL [2], TSQL [12, 13], TempSQL [6], and HSQL [14]

In the late 1970's and early 1980's, two important temporal data models were proposed, namely Ben-Zvi's TRM and Ariav's TOSQL. The designers of these languages did not place much emphasis on valid-time selection and projection; rather, they were interested in specifying the data model. As a result, the support for valid-time selection and projection is very limited in these languages. In the late 1980's and early 1990's, TSQL and HSQL were proposed. These languages have a clearer syntax than did their predecessors, and the the support for valid-time selection and projection is improved. Parallel to these efforts, temporal query languages based on Quel were also being developed. These include TQuel, HQuel, and HTQUEL. Where TQuel is based on 1NF temporal relations, the other two languages are based on non-1NF relations. Therefore, the syntax of HQuel and HTQUEL is very different from that of conventional Quel.

Our objective is to analyze the mechanisms for valid-time selection and projection in existing temporal query languages. Such an analysis is a suitable foundation for designing valid-time selection and projection components of the TSQL2 query language that are powerful and syntactically clear and thus make it easy to formulate and understand queries.

Before we proceed by surveying valid-time selection and projection in nine existing temporal query languages, we first define terminology that is useful when exploring the commonalities and differences between the languages. The language surveyed first is LEGOL 2.0, a pioneer temporal query language. Next, the four extensions to SQL are reviewed in chronological order. Then the three extensions to Quel are examined, also chronologically. A final section contains a summary.

In examples throughout the chapter, we use an employee relation, EMPLOYEE = (NAME, DEPT, POSITION), and the following natural-language query (in addition to other queries) is formulated in each of the covered languages.

Q1. List all of the employees who worked during all of 1990.

Also, the part of each language that relates to valid-time selection and projection is described precisely by means of a BNF syntax. As most of the languages do not define the temporal constants clearly, we use the same format, MM.DD.YY, for all languages. Coverage of temporal constants may be found in [17, Chapter 8].

2.1 Terminology

Below, we define terms that are frequently used in the discussions of the existing language proposals. For clarity, we will use these definitions for all languages, even in situations where other terms were originally used.

Timestamp referencing This term denotes the method provided by the query language for refering to the timestamp values of tuples in queries (e.g., in temporal predicates and target lists).

Event extraction An interval consists of a starting event and an ending event. In temporal queries, it is often convenient to reference (e.g., for comparison) the events of intervals. Event extraction denotes the extraction of the delimiters of intervals.

Interval constructor As the opposite to event extraction, an interval constructor is an operator that creates an interval from events or intervals. One useful constructor takes two events as operands and constructs the interval with these two events as delimiters. Other constructors create new intervals from old intervals by intersection, extension, and union.

Event and interval comparison predicates Temporal query languages provide varying sets of built-in predicates for the comparison of temporal values. We distinguish between predicates for the comparison of events and intervals. While some languages provide separate predicates for the two types of objects, other languages utilize the same predicates.

Temporal predicates By this we mean any boolean expression with comparisons involving time values. Temporal predicates are used for selecting tuples based on their timestamps.

As both events and intervals are present in temporal relations, temporal predicates may involve both types of timestamps. Events are totally ordered and may be treated similarly to integers and reals. Thus, conventional arithmetic comparison operators and logical operators are sufficient for event comparison. Regarding intervals, the comparison operators of Allen have proved to be complete [1]. Note that each of these operators can be simulated with event comparison operators, logical operators, and event extraction mechanisms. (Figure 1 shows the complete set of interval relations defined by Allen.)

Thus, languages that support event time comparison and event extraction have the same expressive power as languages that support interval comparison. While

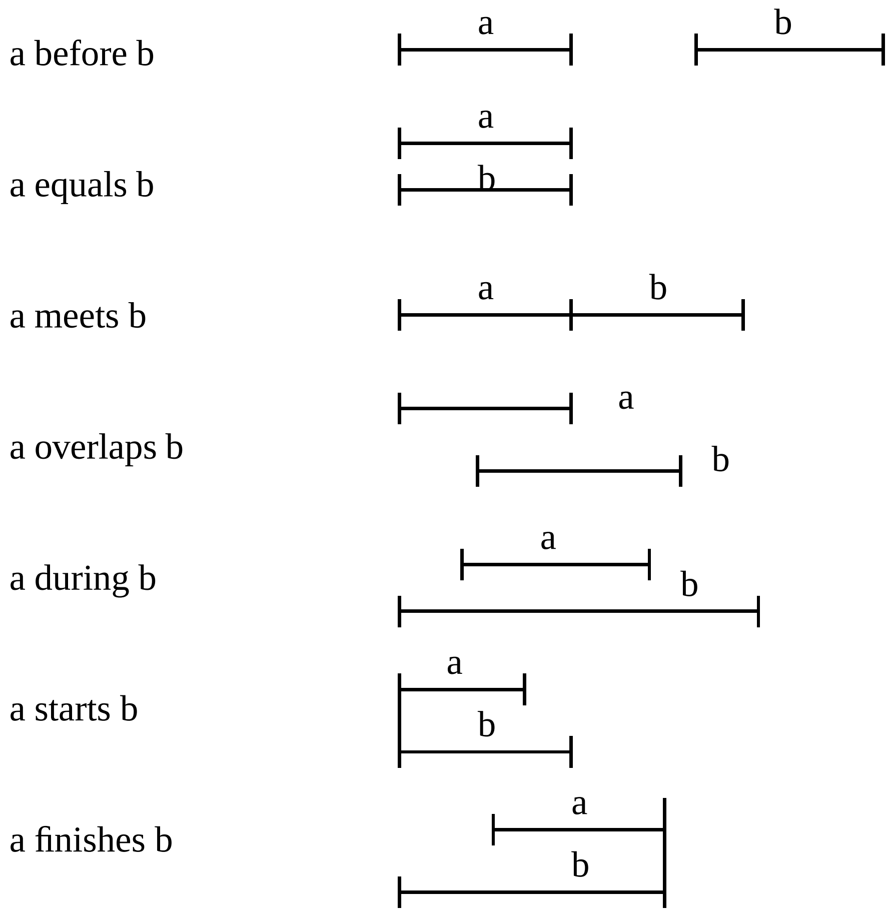

Figure 1: Allen's interval comparison operators

interval comparison operators thus do not increase the expressive power of a query language, such operators do improve the readability of queries.

Valid timeslice This operator retrieves the tuples from the argument relation which are valid during a specified interval or on a time point. Conceptually, it is like "cutting" a slice from a relation.

Temporal ordering A group of timestamped tuples may be numbered according to their timestamp order. By means of temporal ordering, tuples with particular numbers may be selected.

2.2 LEGOL 2.0

Legol 2.0 was the first relational query language to support temporal queries [9]. This language supports tuple-timestamped valid-time relations. Specifically, tuples are assigned two implicit valid-time attributes, `start` and `stop`. These attributes are accessed by the various set-theoretic, temporal, and comparison operators. The query language is closed: The results of queries are valid-time relations. As a result, queries may be nested arbitrarily.

Legol 2.0 differs from other languages in that it is a rule-based, procedural query language. As our interest is declarative query languages, we examine Legol 2.0 only briefly.

In addition to the conventional set-based operations, Legol 2.0 introduces several temporal operations, including valid-time intersection, one-sided valid-time intersection, valid-time union, valid-time difference, and valid-time set membership. The semantics of these operators were explained by examples.

Next, temporal projection is implicitly embedded in the operators. For example, the valid-time intersection operation, implements intersection of timestamps, i.e., the valid time of an output tuple is the intersection of the valid times of the two input tuples. The semantics of these set operations is not clearly implied by the syntax (shown in Figure 2). As a result, it is hard for users to understand queries by simply reading them.

Tuple-variable names are used for implicitly referencing the timestamps of the tuples, and starting and ending events of intervals may be retrieved by the functions, `start of` and `end of`. The arithmetic comparison operators ($>$, $>=$, $<$, $<=$) are applicable also to events. While both event extraction and comparison are supported, Legol 2.0 lacks interval comparison operators and boolean operators. Temporal ordering functions, such as `first, last, current and past`, are provided.

The sample query Q1 is expressed as follows.

```
EMP1990() ⇐ [ start of EMPLOYEE(NAME) < 12.31.90]
             while [ end of EMPLOYEE(NAME) > 1.1.1990]
```

In Legol 2.0, valid-time projection is built-into the operators. Some operators compute new timestamps by intersecting existing timestamps. Other operators define valid-time projection differently, the particular choice depending on the semantics of the operators.

2.3 Ben-Zvi's TRM

Ben-Zvi was the first person to propose a temporal data model with three different times, termed *effective time, registration time,* and *deletion time* [3]. Tuples have five implicit timestamps. The effective time is similar to valid time, and like Legol 2.0, tuples have two of these. There are two registration times, indicating when the valid-time start and the valid-time end were recorded. The single deletion time indicates when a tuple is logically deleted. Together, the registration times and the deletion time provide support for transaction time. A relation of tuples with these temporal attributes is termed a *Time Relation*.

```
<source>               ::=  <expression> {<operator> <expression>}
<expression>           ::=  <simple expression> | “[" <source> “]"
<simple expression>    ::=  <reference> | <literal> | <function call>
<reference>            ::=  <entity reference> | <attribute reference>
<entity reference>     ::=  <entity label> “(" <identifier list> “)"
<entity label>         ::=  <alphabetic string>
<attribute reference>  ::=  <attribute label> of <entity reference>
<attribute label>      ::=  <identifier label> | start | end
<identifier list>      ::=  <identifier element> {“," <identifier element> }*
<identifier element>   ::=  <identifier label> | <literal> | -
<identifier label>     ::=  <alphabetic string> | <non-negative integer>
<literal>              ::=  <quoted string> | <numeric constant>
<function call>        ::=  <function name> [ for <control list>]
                                      of <expression>
<control list>         ::=  <attribute label> {“," <attribute label> }*
<function name>        ::=  <select> | <aggregate> | <value>
<select>               ::=  max | min | highest | lowest |first
                                 | last | current | past
<aggregate>            ::=  sum | number | accumulate|count
                                 | whenever
<value>                ::=  duration | today
<operator>             ::=  <setop> | <timeop> | <arithmeticop>
                                 | <comparison>
<setop>                ::=  union | is | isnot
<timeop>               ::=  while | while not | or while|during
                                 | since | until
<arithmeticop>         ::=  + | - | * | /
<comparisonop>         ::=  = | ≠ | > | < | >= | <=
```

Figure 2: BNF syntax of Legol 2.0

Temporal selection is supported by the `TIME-VIEW` operator and a few simple, built-in temporal predicates. The `TIME-VIEW` operator produces a *snapshot relation*, i.e., a relation without timestamps [16]. The operator accepts two parameters, an `E-TIME` and an `AS-OF` time. The syntax is given as follows (defaults are underlined).

$$\texttt{TIME-VIEW}\left[\texttt{E}-\texttt{TIME}=\left\{\begin{array}{c}\underline{\texttt{NOW}}\\ \texttt{CURRENT}\\ \langle\text{temporal constant}\rangle\end{array}\right\}\right]$$
$$\left[\texttt{AS}-\texttt{OF}=\left\{\begin{array}{c}\texttt{NOW}\\ \underline{\texttt{CURRENT}}\\ \langle\text{temporal constant}\rangle\end{array}\right\}\right]$$

To compute the resulting snapshot relation, the argument relation is first (transaction) time-sliced `AS-OF` some time in the past. Then the intermediate relation is (valid) time-sliced as of the `E-TIME`. The result is the tuples that were effective at `E-TIME`, were registered before the `AS-OF` time, and not deleted before the `AS-OF` time. Because the resulting relation is a non-temporal relation, valid-time projection is not defined in TRM.

TRM provides a limited set of event-comparison operators which may be used in selection predicates.

```
WHERE <qualifier> (<field name>) <comp-operator>
      <temporal constant>
<qualifier> ::=E-START | E-END | R-START | R-END
<comp-operators> ::=< | <= | = | != | >= | >
```

No interval-comparison operators exist, and a temporal predicate is restricted to compare the start of or end of valid or registration times with temporal constants only; there is no way to compare two time variables. For example, a join that compares the start times of two relations can not be expressed in this language. Following is the sample query for Q1 expressed in TRM.

```
SELECT NAME
FROM EMPLOYEE
WHERE E-START(NAME) < 12.31.1990
   AND E-END(NAME) > 1.1.1990
```

The temporal selection predicate in this query simply tests whether the valid times of employee tuples cover the year 1990, but a long and unclear predicate is necessary because interval comparison operators are not available.

```
<temp-spec>       ::=<temp-period> <time-dimension>
<temp-period>     ::=AT [ PRESENT | <temporal const>]
                   |  DURING “("<temporal const> “-" <temporal const> “)"
                   |  BEFORE <temporal const>
                   |  AFTER <temporal const>
                   |  WHILE <selection expr>
                      DURING [ “(" -∞ “-" +∞“)"
                               |  “("<temporal const> “-"
                                   <temporal const>“)" ]
<time-dimension>::=ALONG RT | ALONG <tsa>
```

Figure 3: BNF syntax for constructs in TOSQL related to timeslice

2.4 TOSQL

TOSQL is based on tuple timestamped *transaction-time* relations [2]. In this model, tuples are automatically assigned a *recording time*, abbreviated RT (i.e., a transaction time, the same as registration time in TRM). To obtain additional temporal support, it is possible to define so-called *Time-Related Attributes* (TRAs). Such attributes record time-varying data, for example the times when equipment malfunctions occurred. TRAs that satisfy the *finality property* are termed *Timestamp Attributes* (TSAs). This property is satisfied if at any give time, there exists one and only one value set for each object in the relation. With TSAs, it is possible to record, the valid times of tuples as well as other times. When a TSA is introduced into a relation, its dimensionality is increased by one.

In TOSQL, timeslice may be specified in five ways, each of which are shown in Figure 3. These operators serve as both temporal selection and projection operators for RT (the default) as well as existing TSAs. Temporal projection is computed as the intersection of the period of time given in the timeslice clause and the intervals of selected tuples; no alternatives exist for specifying the timestamps of selected tuples.

The sample query for Q1 is expressed as follows.

```
SELECT F1.NAME
FROM EMP F1
DURING [1.1.1990 - 12.31.1990]
```

Here, the operator `DURING` retrieves all of the tuples which are recorded in 1990, and the `SELECT` operator outputs the employees’ names.

The functionality of these five clauses are similar. All of them, except the `WHILE` clause, can be simulated by the `DURING` clause. Even though the `WHILE` clause does take a selection predicate as argument, it is different from the `WHERE` clause. It singles out periods of time in which tuples satisfy the selection predicate. It's effect may be simulated by a combination of a temporal predicate and the `WHERE` clause. This is illustrated in the following two queries.

Q2. List all the employees who work in the company at some time when Tom is in the toy department.

```
SELECT E.NAME
FROM EMPLOYEE E
WHILE E.NAME = 'Tom' AND E.DEPT = 'toy'
```

The same query, now formulated in TSQL (described in the next section), is expressed as follows.

```
SELECT E1.NAME
FROM EMPLOYEE E1, EMPLOYEE E2
WHERE E2.NAME = 'Tom' AND E2.DEPT = 'toy'
WHEN E1.INTERVAL OVERLAP E2.INTERVAL
```

2.5 TSQL

Navathe's *temporal relational model* supports tuple timestamping for valid time by attaching two mandatory timestamp attributes, *Time-start* (Ts) and *Time-end* (Te) to every time-varying relational schema [10, 12, 13]. These timestamp attributes correspond to the lower and upper bounds of time intervals in which tuples are continuously valid.

Informally speaking, attributes are not allowed to have multiple values at any particular instant of time in relation instances in this model. As a result, the time invariant key (i.e., the primary key of the corresponding snapshot relation schema, abbreviated TIK) together with either Ts or Te defines a candidate key for a temporal relation. Next, *value-equivalent* tuples (i.e., tuples with identical non-timestamp attribute values) are required to be coalesced. Coalescing affects the facility for temporal ordering of tuples, and requiring it improves the utility of this facility.

Most temporal operations are supported in TSQL, as are the properties defined in Section 2.1. We will describe these operations next, starting with temporal predicates and continuing with temporal projection, temporal ordering, and timeslice.

In TSQL, a temporal selection predicate is specified in the when clause which is a temporal analogue to SQL's where clause. The clause consists of a new keyword `WHEN` followed by a temporal boolean expression (see Figure 4 for the BNF syntax) which in turn consists of temporal predicates and logical operators.

The predicates consist of temporal expressions, which return intervals or events, and temporal comparison operators. In temporal expressions, three postfix operators are used for referencing the timestamps of tuples. The event extrac-

```
<time-slice-clause> ::=  TIME-SLICE <tem-const>
<when-claus>        ::=  WHEN <boolean>
<boolean>           ::=  <bool-term> | <boolean> OR <bool-term>
<bool-term>         ::=  <bool-fact> | <bool-term> AND <bool-fact>
<bool-fact>         ::=  [NOT] <bool-prim>
<bool-prim>         ::=  <predicate> | ( <boolean> )
<predicate>         ::=  <expr> <tem-comp-op> <expr>
<expr>              ::=  <tem-seq> <ts-var> [<bf-af> <tem-seq> BREAK]
                         | [<tem-seq>] <ts-var> | <tem-const>
<tem-const>         ::=  <t-term> | “[" <t-term> “," <t-term> “]"
<t-term>            ::=  <t-fac> <add-op> <number>
<t-fac>             ::=  <t-s-v> | NOW | <number> <t-s-f>
<tem-com-op>        ::=  OVERLAP | EQUIVALENT | FOLLOWS
                         | BEFORE | AFTER | ADJACENT | PRECEDES
                         | DURING
<tem-seq>           ::=  FIRST | SECOND | THIRD | Nth | LAST
<ts-var>            ::=  <ts-attr> | <ts-attr-fid>
<ts-attr>           ::=  [<rel-name>“."] <time-st>
<ts-attr-fid>       ::=  [<rel-name>“."] <time-st>“."<t-s-f>
<time-st>           ::=  TIME-START | TIME-END
<bf-af>             ::=  BEFORE | AFTER
<t-s-v> : refer to value appearing under Ts(time start) or Te(time end).
<t-s-f> : temporal fields( e.g., year, month, day)
```

Figure 4: BNF of temporal selection in TSQL

tion operators `.TIME-START` and `.TIME-END` return the start of and end of time intervals, respectively. The interval extraction operator, `.INTERVAL`, returns the entire interval.

Eight comparison operators all of which apply to interval arguments are included, namely `BEFORE`, `AFTER`, `EQUIVALENT`, `PRECEDES`, `FOLLOWS`, `OVERLAP`, `DURING`, and `ADJACENT`. Events are treated as degenerate intervals, so that the comparison operators also apply to events. Based on Allen's interval relationships, Table 1 provides an overview of the interval comparison operators of TSQL, HSQL, and TQuel The table shows that TSQL has operators that correspond to each of Allen's interval relationships, with the exception of start and finish (which are rarely used). The table also indicates that the three languages are all complete with respect to interval comparison. The semantics of the eight operators in TSQL is clear, so a TSQL temporal predicate is easier to understand and construct than predicates in the languages described in the previous three sections.

The sample query Q1 is expressed as follows in TSQL.

```
SELECT F1.NAME
FROM EMP F1
WHEN F1.INTERVAL CONTAINS "1990"
```

Valid-time projection is specified in the target list, i.e., in the `SELECT` clause, with the format of a relation quantifier name followed by `.TIME-START`, `.TIME-END`, or `.INTERVAL`. If there is more than one relation included in the query, the quantifier name is necessary in order to indicate which timestamp to retrieve; otherwise, the quantifier name can be omitted. An interval constructor, `INTER`, which returns the intersection of two intervals can be used in the target list when specifying the timestamps of the resulting relation. This is illustrated in query Q3 below.

Q3. For all employees that worked in the same department as Mike and worked throughout 1990, find the time when they worked with Mike.

```
SELECT F1.NAME, (F1 INTER F2).TIME-START,
    (F1 INTER F2).TIME-END
FROM EMPLOYEE F1, EMPLOYEE F2
WHERE F2.NAME = 'Mike' AND F1.DEPT = F2.DEPT
WHEN F1.INTERVAL OVERLAP 1990
    AND F2.INTERVAL OVERLAP 1990
```

According to the BNF syntax, the `.INTERVAL` in the `WHEN` clause cannot be omitted for indicating a timestamp. However, the tuple variable name can represent the timestamp in the `SELECT` clause (not shown in this chapter). This is an inconsistency in the design—there is no logical reason for this difference.

Temporal ordering is well supported in TSQL because users can retrieve any version of an entry in its global or local ordering (defined later). No other languages support both global and local ordering. The underlying data model ensures that in any relation instance, no two tuples with the same TIK values have overlapping intervals. Thus, tuples can be grouped based on their TIK value and then be assigned unique order numbers based on either the start or the end times of their intervals. This is termed the *global ordering*. It is possible for the intervals in a group to be non-consecutive, i.e., there may be *breaks*. Such breaks may also be ordered and assigned order numbers. The breaks introduce a sub-partitioning of the tuples in a group. The tuples in the sub-groups may also be ordered—this is termed the *local ordering*.

To specify global temporal ordering in TSQL (see Figure 4 for the syntax), an order number,e.g., `FIRST` or `SECOND`, is placed before an attribute variable. Considering local ordering, a desired sub-group can be specified by indicating a break number, e.g., `AFTER FIRST BREAK`, behind an attribute variable. Some examples describe the use of global and local temporal ordering.

Allen	TSQL	HSQL	TQuel

`a before b`	`a BEFORE b`
	or
	`b AFTER a`
		`a PRECEDES b`	`a precede b`
			`and`
			`not (end of a equal begin of b)`

`a equals b`	`a EQUIVALENT b`	`a = b`	`a equal b`

`a meets b`	`a PRECEDES b`
	or
	`b FOLLOWS a`
		`a MEETS b`	`end of a equal begin of b`

`a overlaps b`	`a OVERLAP b AND`
	`a.TIME-END PRECEDES b.TIME-END`
		`a OVERLAP b`
		`AND`
		`a.TO < b.TO`
			`a overlap b and`
			`end of a precedes end of b`

`a during b`	`a DURING b`	`b CONTAINS a`	`begin of a overlap b and`
			`end of a overlap b and`
			`not (a equal b)`

`a meets b`
`or`
`b meets a`
	`a ADJACENT b`	`a ADJACENT b`	`end of a equal begin of b`
			`or`
			`end of b equal begin of a`

`a starts b`	`(a.TIME-START EQUIVALENT`
	`b.TIME-START) AND`
	`(a.TIME-END PRECEDES`
	`b.TIME-END)`
		`a.FROM = b.FROM`
		`AND`
		`a.TO < b.TO`
			`begin of a equal begin of b`
			`and`
			`end of a precede end of b`

`a finishes b`	`(a.TIME-START AFTER`
	`b.TIME-START) AND`
	`(a.TIME-END EQUIVALENT`
	`b.TIME-END)`
		`a.FROM > b.FROM`
		`AND`
		`a.TO = b.TO`
			`begin of a precede begin of b`
			`and`
			`end of a equal end of b`

Table 1: Overview of interval comparison operators in TSQL, HSQL, and TQuel

Q4. Find the names of the employees who started in the toy department.

```
SELECT E.NAME
FROM EMPLOYEE E
WHERE FIRST(E.DEPT) = 'TOY'
```

Q5. For employees that joined the company a second time and received a salary exceeding $ 50,000, retrieve the name and the time when they joined.

```
SELECT E.NAME E.TIME-START
FROM EMPLOYEE E
WHERE FIRST(E.SALARY) > 50000 AFTER FIRST
BREAK
```

Q6. Find the start time and salary of employees when they entered the toy department the first time.

```
SELECT E.NAME FIRST(E.SALARY) E.TIME-
START
FROM EMPLOYEE E
WHERE E.DEPT = 'TOY'
```

Time-slice queries are expressed using a special clause, `TIME-SLICE`. In this clause, an interval, constructed using temporal constants enclosed within square brackets, or a time point, defined by a temporal constant, is specified. The temporal constant can be a constant expression thus allowing dynamic timeslices. This property is very useful for a fixed-length timeslice which moves along in the valid-time dimension. An example is given below.

Q7. On a monthly basis, list all employees who worked in the company the last year.

```
SELECT NAME
FROM EMPLOYEE
TIME-SLICE YEAR[NOW-1, NOW]
```

This query can be executed monthly without modification.

2.6 HSQL

As the previous data model, Sarda's HDBMS also supports valid time; however, unlike the data model mentioned previously, HDBMS represent valid time in a valid-time relation as a single non-atomic, implicit attribute [14]. The query language of HDBMS is called HSQL[1]. The valid timestamps of tuples can be either

[1]In another paper, Sarda gave this extension to SQL the name TSQL [15]. We use HSQL because it was used in the most recent paper.

intervals, in *state relations*, or events, in *event relations*. The interval comparison, event comparison, and interval operations are defined for operating on timestamps. The granularity is not fixed in HDBMS where users can define different granularities for each relation. Comparison of events in different granularities is defined in HDBMS by that a coarser event is converted to an interval of finer granularity, then the comparison of the interval and the event of finer granularity is executed. Other operations dealing with granularity, such as expand and coalesce, are also provided. Details of temporal predicates, valid-time projection and timeslice are described in the following.

The functionality of temporal predicates and timeslices in HSQL is similar to that of TSQL. The difference are mainly in syntax and temporal comparison operators.

A simplified BNF is shown in Figure 5 (Since Sarda does not provide any full syntax in his articles, the BNF is derived by us and may not be accurate, but it should be close to HSQL as proposed in Sarda's paper).

```
<timeslice-clause>    ::=  FROM <temporal const>
                           [TOTIME <temporal const>]
<where-clause>        ::=  WHERE <boolean>
<boolean>             ::=  <bool-term> | <boolean> OR <bool-term>
<bool-term>           ::=  <bool-factor> | <bool-term> AND <bool-factor>
<bool-factor>         ::=  [NOT] <bool-prim>
<bool-prim>           ::=  <regular pred> | <temporal pred> | ( <boolean> )
<temporal pred>       ::=  <interval> <interval-comp-op> <interval>
                           | <instant> <inst-comp-op> <instant>
                           | <instant> IN <interval>
<interval>            ::=  <interval-fac> <interval-op> <interval-factor>
                           | <inst-factor> ".." <inst-factor>
<interval-op>         ::=  + | *
<interval-fac>        ::=  <interval-var> | (<interval>)
<interval-var>        ::=  <rel-name>"."INTERVAL
<inst-factor>         ::=  <temp-const> | <rel-name>"."FROM
                           | <rel-name>"."TO
<interval-comp-op>    ::=  PRECEDES | = | MEETS | OVERLAPS
                           | CONTAINS | ADJACENT
<inst-comp-op>        ::=  < | <= | = | <> | >= | >
```

Figure 5: Simplified BNF for HSQL temporal selection

Unlike TSQL, which provides a `when` clause for temporal predicates, HSQL simply adds temporal predicates to the `where` clause. Boolean expressions in the `where` clause are composed of temporal predicates and non-temporal predicates. The event extraction operators are `.FROM` and `.TO` which retrieve the start and end of an interval, respectively. The timestamp-referencing operators are `.INTERVAL` and `.AT`; the `.INTERVAL` represents the interval timestamp of a tuple in a state relation, and the `.AT` indicates the event time associate with a tuple of a event relation. Another difference between temporal predicates in TSQL and HSQL is that HSQL supports two different set of comparison operators for interval time and event time. The arithmetic comparison operators are overloaded for event time. while these are the interval comparison operators: `PRECEDES, =, MEETS, OVERLAPS, CONTAINS`, `ADJACENT`. The only exception is the operator `IN` which tests if an event is contained in an interval. When formulating queries, users of HSQL must pay more attention than must TSQL users in order to choose the right comparison operators for the different temporal variables and constants. Although the set of interval comparison operators is smaller than that of TSQL, the expressive power is the same, see Table 3.

In order to explain how valid-time projection is done in HSQL, we introduce the interval constructors which are defined as follows.

- t_1 `..` t_2: constructs a period of time from t_1 to t_2. It is null if $t_1 >= t_2$.
- p_1 `+` p_2: construct a period equals (min(p_1.`FROM`, p_2.`FROM`), max(p_1.`END`, p_2.`END`)) if p_1 and p_2 overlap; otherwise, it returns a null interval.
- p_1 `*` p_2: return the intersection of p_1 and p_2. If there is no intersection, a null interval is returned.

The valid-time projection is explicitly specified the timestamps in the select clause. The timestamps are either constructed by the interval constructors with timestamps of selected tuples or just the timestamps of selected tuples. If no timestamp is specified in the `select` clause then the result is a snapshot relation. For example, the query "list the employees in toy the department in 1990" can be expressed as follows.

Q8. List the employees in toy department in 1990.

```
FROMTIME 1.1.90 TOTIME 12.31.90
SELECT R.NAME, R.PERIOD * (1.1.90 ..  12.31.90)
FROM EMPLOYEE R
WHERE R.DPET = 'TOY'
```

This is also an example of timeslice. If `TOTIME` is omitted, the default time, `NOW`, is taken. Sample formulations of the query **Q1** are given below. Note that there are two ways to write the query. One uses a temporal predicate; the other uses the timeslice operation.

```
SELECT F1.NAME
FROM EMP F1
WHERE F1 OVERLAP "1990"
```

```
FROMTIME 1:1:1990 TOTIME 12:31:1990
SELECT F1.NAME
FROM EMP F1
```

2.7 TempSQL

Gadia's TempSQL is based on a N1NF relational temporal data model which is attribute-value timestamped [6, 7, 8]. A tuple may have more than one value (time-stamped) for each attribute, but the union of the timestamps in each attribute must be the same throughout the entire tuple, resulting in a homogeneous temporal relation. In their data model, they group a history of an entry into a tuple, instead of storing the history of an entry in multiple tuples as is done in 1NF data models. The following is an example of an attribute timestamped relation.

NAME	SALARY	DEPT
[11,60] John	[11,49] 15K [50,54] 20K [55,60] 25K	[11,44] Toys [45,60] Shoes
[0,44] ∪ [50,NOW] Mary	[0,44] ∪ [50,NOW] 25K	[0,44] ∪ [50,NOW] Credit

In TempSQL, valid-time selection is heavily dependent on *temporal expressions* which are specified in the `WHILE` clause; see Figure 6 for a BNF. Temporal expressions are of the form, ⟦ A ⟧, ⟦ r ⟧, ⟦ AθB ⟧, and ⟦ A θ b ⟧ where A and B are attributes, b is an attribute value and r is a relation. Thus the timestamp referencing is specified as a temporal expression. A temporal expression returns a *temporal element* [5] which is the finite union of intervals where the temporal expression is true. More complex temporal expressions are formed using the operators ∪, ∩, and −. For example, ⟦ SALARY = 20K ⟧ is a temporal expression. If the expression is applied to John's tuple, it would return [50,54].

There are two ways to specify temporal selection in TempSQL, depending on whether the time domain of the resulting relation is restricted to time elements specified in the `WHILE` clause. For example, the following query is interpreted as "list the employees when they worked in the toy department".

```
SELECT *
WHILE ⟦ DEPT = TOYS ⟧
FROM EMP
```

The result is as follows.

NAME	SALARY	DEPT
[11,44] John	[11,44] 15K	[11,44] Toys

A similar query, "list the employees who worked in the toy department," would list any one who had ever worked in the toy department.

```
<select stmt>       ::= <select clause> <while clause> <from clause>
                              <where clause>
<select clause>     ::= select <target list> [:<attr list>]
<while clause>      ::= while <temporal expr>
<from clause>       ::= from <relation list>
<where clause>      ::= where <bool expr>
<relation list>     ::= <relation name> [:<attr list>] [, <relation list>]}*
<temporal expr>     ::= <temporal value>
                    |   ¬ <temporal value>
                    |   <temporal value> <set oper> <temporal value>
                    |   <temporal constant>
<set oper>          ::= ∩ | ∪
<temporal value>    ::= ⟦ <attr name> ⟧
                    |   ⟦ <predicate> ⟧
                    |   ⟦ <select stmt> ⟧
                    |   ⟦ exists <select stmt> ⟧
                    |   ⟦ <attribute> in <select stmt> ⟧
                    |   ⟦ <attribute> not in <select stmt> ⟧
                    |   ⟦ <attribute> <comparison oper> any <select stmt> ⟧
                    |   ⟦ <attribute> <comparison oper> all <select stmt> ⟧
<bool expr>         ::= <bool value>
                    |   ¬ <bool value>
                    |   <bool value> <bool oper> <bool value>
                    |   “(" <bool value> “)"
<bool oper>         ::= and | or
<bool value>        ::= <predicate>
                    |   <temporal value> = <temporal value>
                    |   <temporal value> ≠ ∅
                    |   <temporal value> ⊇ <temporal value>
<predicate>         ::= <attribute> <comparison oper> <attribute>
                    |   <attribute> <comparison oper> <value>
                    |   <value> <comparison oper> <attribute>
```

Figure 6: BNF for temporal selection in TempSQL

```
SELECT *
FROM EMP
WHERE ⟦ DEPT = TOYS ⟧ ≠ ∅
```

The result is as follows.

NAME	SALARY	DEPT
[11,60] John	[11,49] 15K [50,54] 20K [55,60] 25K	[11,44] Toys [45,60] Shoes

In the first query, the time domain of the result is restricted by the temporal expression, ⟦ `DEPT = TOYS` ⟧. The restriction represents a valid-time projection, i.e., valid-time projection is defined by the `WHILE` clause. The timestamp of the result relation is the intersection of the attribute timestamp with the temporal expression. In the second query, there is no restriction on the time domain of the result, so the entire tuple which satisfies the `WHERE` clause is retrieved.

There is no special timeslice operator defined in TempSQL (likewise for temporal ordering). Event extraction could be specified by two functions `firstInstant` and `lastInstant`. But it is not clear whether or not event comparison operators are supported.

2.8 TQuel

Snodgrass's TQuel is based on a tuple timestamped, temporal data model supporting both transaction time and valid time. Unlike TRM, where the result of a query is a snapshot relation, the result of a TQuel query is a valid-time relation, so the data model is consistent. Like HDBMS, TQuel distinguish between event relations and interval relations ("state relation" in HDBMS); however, in TQuel the temporal comparison operators are overloaded for both interval and event time.

TQuel adds a new clause, `when`, for valid-time selection.

Although not an extension of SQL, the BNF of the `when` clause as shown in Figure 7 still gives a good illustration of how valid-time selection may be done syntactically in TSQL2. The temporal predicates can be partitioned into two groups: the first is predicates that consist of interval expressions, event expressions, and the temporal predicate operators, `overlap, precede,` and `equal`, which are overloaded for both interval and event time comparison; the second group is the boolean expressions that consist of predicates of the first group and logical operators.

The timestamps of tuples are referenced by tuple-variable names and the event extraction operators, `begin of` and `end of`. There are also two interval constructors, `overlap` and `extend`, for constructing temporal expression. The `overlap` operator returns an intersection of time when both arguments are valid.

```
<when clause>      ::=  when <temporal pred>
<temporal pred>    ::=  <ei-expression> precede <ei-expression>
                   |    <ei-expression> overlap <ei-expression>
                   |    <ei-expression> equal <ei-expression>
                   |    <temporal pred> and <temporal pred>
                   |    <temporal pred> or <temporal pred>
                   |    (<temporal pred>)
                   |    not <temporal pred>
<ei-expression>    ::=  <e-expression>
                   |    <i-expression>
<e-expression>     ::=  <event element>
                   |    begin of <ei-expression>
                   |    end of <ei-expression>
                   |    (<e-expression>)
<i-expression>     ::=  <interval element>
                   |    <ei-expression> overlap <ei-expression>
                   |    <ei-expression> extend <ei-expression>
                   |    (<i-expression>)
<valid-clause>     ::=  valid from <e-expression> to <e-expression>
                   |    valid at <e-expression>
```

Figure 7: BNF for temporal selection in TQuel

On the other hand, the `extend` operator is more like temporal union, in that it returns the points in time when either of the arguments are valid. These operators can be specified in event expressions and interval expressions for constructing a new event or interval. The sample query **Q1** is expressed as follows .

```
range of f1 is Emp
retrieve into r(name = f1.name)
when f1 overlap "1990"
```

Although, the set of comparison operators has three operators only, together with the functions of timestamp referencing and event extraction, the set has the same expressive power as the other languages (see Tables 2–5). However, long temporal expressions in TQuel may be used to express similar predicates in other languages. For example, in Table 2, the operator `during` in TSQL is expressed as the conjunction of three predicates in TQuel.

The `valid` clause is used for specifying valid-time projection, which in TQuel can be any period of time. If the derived relation is to be an event relation, the `valid at` <t1> variant specifies a single timestamp. The other variant of the valid clause, `valid from` <t1> to <t2>, specifies an interval timestamp and is used when the derived relation is to be an interval relation. Both of t1 and t2 can be derived from event expressions given in previous paragraphs. The query **Q3** is expressed in the following query which contains timestamp referencing, event extraction, interval constructor, valid clause, and temporal predicate.

```
range of e1 is employee
range of e2 is employee
retrieve into r(name= e1.name)
valid from begin of (e1 overlap e2) to
            end of (e1 overlap e2)
where e2.name = "Mike" and e1.dept = e2.dept
when e1 overlap 1990 and e2 overlap 1990
```

2.9 HQuel

Tansel's HQuel is based on a valid-time relational data model where the attributes are timestamped, leading to non-first normal form relations (N1NF) [18]. The data model has four different kind of attributes which are elementary (atomic), set-valued, triplet-valued , and set triplet-valued attributes. The elementary and set-valued attributes are non-time-varying attributes; the other two are time-varying attributes. The values of triplet-valued attributes are triplets containing an element from the attribute's value domain and the boundary points of the interval of existence, while the value of set triplet-value attribute is a set of such triplets. The relation resulting from a query retains these four kinds of attributes depending on how the query is specified.

Because of the varieties of attributes, HQuel supports methods for referencing members of a set, for indicating the timestamps and value of a triplet, and for comparing attribute values. Set members are referenced by a new range variable which ranges over the elements of a set. We describe this by an example. Let t be a regular range variable, as in Quel, and `*A` be a set-valued attribute. `Range of s is t.*A` declares that s ranges over the elements of attribute `A` for each tuple in relation `t`. Assuming that `$B` is a triplet-valued attribute then `t.$B(V), t.$B(L), t.$B(U)`, and `t.$B(T)` represent the value, lower-bound(time start), upper-bound(time end) and interval of a triplet valued attribute, `t.$B`.

The new comparison operators include set comparison operators ($\rho \in \{=, \neq, \supset, \supseteq\}$), a set membership operator ($\in$), and temporal comparison operators. An interval comparison expression is of the form $x\theta y\rho z$ where x, y and z are time intervals, and $\theta \in \{\cap, \cup, -\}$. For example, "`t.$B(T)` $\cap$ `t.$C(T)` $\neq \emptyset$" means that the intersection of the time of t[$B] (denotes the set of triplet-valued attributes B in relation t) and the time of t[$C] is not empty, or simply, intervals of B and C overlap. This is not intuitive to users, since time is treated as a set of discrete points instead of a contiguous time line. Not only that, the interval comparison expression does not have the same expressive power as Allen's definition; for example, there is no way to express "before" in HQuel.

For event comparison, the arithmetic comparison operators ($<, <=, =, \neq, >=, >$) are used to express an event-time predicate which contains event-time variables or temporal constants. For example, `t.$B(L) > 1/91` means the time of triplet-valued attribute B start after January 1991. Although, the interval comparison operators are not complete in HQuel, the event-time comparison operators and timestamp retrieval operators have the same expressive power as other languages in Tables 2–5. The sample query **Q1** is expressed as follows, assuming that `position` is a set triplet-valued attribute.

```
range of f1 is Emp
range of p1 is f1.*$position
retrieve into r(f1.name)
where p1.$(T) ∩ "1990" != ∅
```

In HQuel, the valid-time projection is defined explicitly in the retrieve clause. If an attribute of a result relation is supposed to be a time-varying attribute, a triplet-valued attribute is specified in the `retrieve` clause. The set operators can also be used in valid-time projection for constructing timestamps in HQuel. The following is an example.

Q9. What was Tom's position when he worked for the toy department, and when was it?

```
range of e is Emp
range of s is e.*$position
range of d is e.*$dept
retrieve into Tompos($Tpos =
        <s.$position(T) ∩ d.$dept, s.$position(V)>)
where e.ename = Tom and d.$dept(V) = Toy
    and s.$position(T) ∩ d.$dept ≠ ∅
```

2.10 HTQUEL

Gadia's HTQUEL is based on the same data model for TempSQL, but HTQUEL is an extension of Quel. We will not discuss the data model again, but talk only about the language features. The function for timestamp referencing is `tdom` which takes an attribute as argument and returns a *temporal element*, a finite union of disjoint intervals. Because the timestamp of an attribute is a set of intervals, the functions for event extraction are applied on temporal elements. `Firstinstant(v)` and `Lastinstant(v)` return the two boundary time points of the temporal element,

v. `Nextinstant(v,t)` takes an temporal element, v, and a instant, t, then returns an instant after t; `Previousinstant` is similar, but returns an instant before t. The functionalities of the functions, `Firstinterval, Lastinterval, Nextinterval, and Previousinterval`, are similar to the event extraction functions, but these four operations return intervals. HTQuel supports timeslice via the `during` clause. Most of the other time-related operations are not defined clearly in the papers (e.g., such as event comparison, interval comparison, and temporal projection). We can thus not compare the expressive power with that of other languages. The sample query **Q1** is expressed as follows.

```
range of f1 is Emp
retrieve into r(name = f1.name)
during [1990]
```

2.11 Summary

In this section, we summarize the languages with respect to valid-time selection and projection, in two tables (Table 2 and Table 3), and we evaluate the query languages on ten properties (defined below). The first row in the tables contain the references to each language and the remaining rows each relate to a property. The first three properties, timestamp referencing, event extraction and event comparison, are essential to temporal predicates; the interval comparison and time slice improve syntactic convenience, but not the expressive power. An *event constructor* is a function which takes events as arguments and returns an event. An example of this is `first` which returns an oldest event from the arguments. With event constructors and interval constructors, we can write queries with advanced event or interval expressions. The remaining 3 properties are listed for comparison; the language that supports some of these properties is not necessarily better than other languages.

In the summary table, we show the keywords of a language, if the language supports the property. Otherwise, "*No*" denotes not supporting a property in a certain language, and "*Unclear*" denotes that the original papers were unclear with respect to the property. In the previous subsections, we have discussed how each aspect is done in each language. We now describe these properties briefly and point out some interesting facts.

- *timestamp referencing*: How can timestamps of tuples be referenced? Generally, there are two ways: by an operator or through the tuple variable name. TSQL and HSQL use operators; Legol 2.0 and TQuel use tuple-variable names to denote the timestamp of a tuple.
- *Event extraction*: What are the functions or operators for extracting the delimiters of an interval. The operators fall into two categories: prefix operators

or postfix operators. The prefix operators are more like function calls that take interval expression as arguments. Postfix operators look like attribute and cannot operate on expressions, which limits their power. Because the data model of HQuel and HTQUEL are N1NF, they both use special ways to extract events.

- *Event time comparison*: What are the operators for comparing events? All of the operators in these languages are infix and take event time constants and event time expressions as arguments. Half of the languages borrow the arithmetic comparison operators while TSQL and TQuel use keywords as operators.
- *Interval comparison*: Are there operators for comparing intervals? Three languages—TSQL, HSQL, and TQuel—support sets of helpful interval comparison operators, while other languages provide either no support or partial support.
- *Event constructor*: Is there any constructor that takes event time as arguments and returns an event? None of the nine languages support event constructors; however, we believe that event constructors are useful. This is the reason why event constructors are included in the summary.
- *Interval constructor*: Is there any constructor that operates on intervals or events and returns an interval? The common interval constructors are intersect and union(extend), but the definition and syntax may vary from language to language.
- *timeslice construct*: Is a timeslice construct supported in the language? Generally, the timeslice operation can be replaced by a simple temporal predicate (shown in next section). "*None*" denotes that no special construct for timeslice exists. Otherwise, we give the keywords.
- *Temporal ordering*: Is there a way to retrieve a tuple from a group according to its temporal ordering in the group? Most query languages provide functions for the first and last in temporal order, except TSQL which provides a set of functions for retrieving any version of an entity.
- *New clause or operators for temporal selection*: Is a new clause defined for temporal predicates in the language? Most of the languages do not introduce a new clause for temporal predicates—TQuel, TSQL, and Legol 2.0 are exceptions. Legol 2.0 is a procedural query language which needs many operators to accomplish the desired functionalities.
- *Valid-time projection*: How are the timestamps defined for the resulting relation? There are three possibilities: the valid-time projection can be defined in the target list, it can be defined in a new clause, or it can be mixed with valid-time selection operators. TSQL, HSQL, and HQuel employ the target list, and TQuel uses the `valid` clause. Both TOSQL and Legol 2.0 mix

Language	TRM	TOSQL	TSQL
Reference	[3]	[2]	[11] [12]
Timestamp Referencing	*No*	*No*	*t*`.INTERVAL` *or* *tuple variable name* *(in* `SELECT` *clause)*

Event Extraction	`E-START(`*attr*`)`
	`E-END(`*attr*`)`
	`R-START(`*attr*`)`
	`R-END(`*attr*`)`
		No	*t*`.TIME-START`
			t`.TIME-END`

Event time Comparison	`<, =, >`	*No*	*same as below*

Interval Comparison	*No*	*No*	`BEFORE`
			`AFTER`
			`EQUIVALENT`
			`PRECEDES`
			`FOLLOWS`
			`OVERLAP`
			`DURING`
			`ADJACENT`

Event Constructors	*No*	*No*	*No*
Interval Constructors	*No*	*No*	`INTER`

Timeslice Construct	`TIME-VIEW`	`AT`
		`DURING`
		`BEFORE`
		`AFTER`
		`WHILE`
			`TIME-SLICE`

Temporal Ordering	`T-FIRST`
	`T-LAST`
		No	`FIRST`
			`SECOND`
			`THIRD`
			`NTH`
			`LAST`

New clause or Operators for Valid-time Selection	*No*	*No*	`WHEN`
Temporal Valid-time	*No*	*Mixed with timeslice operators*	`SELECT` *clause* *vs* *WHEN clause*

Table 2: Summary—Part I

Language	HSQL	TempSQL	Legol 2.0
Reference	[15] [14]	[6]	[9]
Timestamp Referencing	*t*`.INTERVAL` *t*`.AT`	⟦ *relation name* ⟧	*tuple variable*

Event Extraction	*t*`.FROM`
	t`.TO`
		`firstInstant`⟦*u*⟧
		`lastInstant`⟦*u*⟧
			`start of(`*exp*`)`
			`end of(`*exp*`)`

Event time Comparison	`<, =,>`	*unclear*	`<, =, >`

Interval Comparison	`PRECEDES`
	`=`
	`MEETS`
	`OVERLAPS`
	`CONTAINS`
	`ADJACENT`
		$\cap, \cup, \supseteq$	*Partial*

Event Constructors	*No*	*unclear*	*No*
Interval Constructors	t_1 `..` t_2 $*$ $+$	⟦ *temp expr*⟧	*No*
timeslice Construct	`FROMTIME` *t*1 `TOTIME` *t*2	*None*	*None*

Temporal Ordering	`FIRST`
	`LAST`
		No	`first`
			`last`
			`current`
			`past`

New clause or Operators for Valid-time Selection	*No*	`WHILE`	`while`
			`since`
			`until`
			`during`
			`while not`
			`or while`
			`union`
			`is`
			`is not`

Temporal Valid-time	`SELECT` *clause* *vs* `WHERE` *clause*	*WHILE clause* *vs* `WHERE` *clause*	*Mixed with temporal selection*

Table 3: Summary—Part II

Language	TQuel	HQuel
Reference	[16] [Snodgrass 1987]	[18]
Timestamp Referencing	*tuple variable*	`$`*attr*`(T)`

Event Extraction	`begin of` *exp*
	`end of` *exp*
		`$`*attr*`(L)`
		`$`*attr*`(U)`

Event time Comparison	`precede` `equal`	`<, =, >`

Interval Comparison	`precede`
	`equal`
	`overlap`
		Partial
		$(=, \neq, \subset, \subseteq)$

Event Constructors	*No*	*No*
Interval Constructors	`overlap` `extend`	*No*
Timeslice Construct	*None*	*None*
Temporal Ordering	*No*	*No*
New clause or Operators for Valid-time Selection	`when`	*No*
Valid-time Projection	*valid clause* *vs* *when clause*	*From target list* *vs* *where clause*

Table 4: Summary—Part III

Language	HTQUEL
Reference	[7]
Timestamp Referencing	`tdom(`attr`)`

Event Extraction	`Firstinstant(`ν`)`
	`Lastinstant(`ν`)`
	`Nextinstant(`ν`,`i`)`
	`Previousinstant(`ν`,`i`)`

Event time Comparison	*Unclear*
Interval Comparison	*Unclear*
Event Constructors	*No*

Interval Constructors	`Firstinterval(`ν`)`
	`Lastinterval(`ν`)`
	`Nextinterval(`ν`,`i`)`
	`Previousinterval(`ν`,`i`)`

Timeslice Construct	`During`
Temporal Ordering	*No*
New clause or Operators for Valid-time Selection	*No*
Valid-time Projection	*Unclear*

Table 5: Summary—Part IV

valid-time projection with valid-time selection.

3 Design Criteria

As a guide for making appropriate design decisions, we present some language design criteria. These criteria are *expressive power, consistency, clarity, minimality, orthogonality,* and *independence.* Initially, each criterion is described. Then the interactions among the criteria are exemplified.

Expressive Power This criterion indicates that the language must exhibit a functionality that makes it suitable for its intended applications and does not impose undesirable restrictions on the queries that may be formulated.

This does not mean that providing a lot of operators and functions is necessarily better than a more restricted set. For example, this criterion has implications for TSQL2's choice of comparison operators that involve valid time.

Consistency For the task at hand, this criterion has at least four implications. First, the design must be consistent with the syntax for user-defined time support in TSQL2. For example, it should use the formats for temporal constants adopted there (see [17, Chapter 8]). Second, the design should be upward compatible with SQL2. This indicates that defaults should be chosen carefully. Third, the design should be consistent with the designs of other aspects of TSQL2. Fourth, the syntax should be internally consistent. For instance, mixing postfix and prefix operators is not considered a good design.

Clarity The syntax should clearly reflect the semantics of the language. This aids in formulating and understanding queries. Applications of the principle include the meaningful naming of operators, a proper choice of clauses (to obtain well-structured queries), and a consistent naming style. As a specific example, inclusion of a period comparison operator such as `OVERLAPS` increases readability when compared with an equivalent predicate based on event extraction and event comparison operators.

Minimality The principle of minimality indicates that as few as possible new reserved words and clauses should be introduced and added to those already present in SQL2. It also indicates that new operators should not be included if they duplicate the functionality already provided by existing operators. This is intended to ensure that users will not be unnecessarily burdened by a large set of operators.

Orthogonality It should be possible to freely combine query language constructs that are semantically independent. The Zero–One-Infinity principle may be seen as a more specific design criterion. This criterion states that the only reasonable numbers in a design are zero, one and infinity and that other numbers are unintuitive to users. For example, restricting the number of tuple variables that may be declared in a query to another number (e.g., 15) appears to have no logical explanation and is difficult to remember.

Independence Obeying this criterion ensures that each function is accomplished in only one way. Designing functions to be independent and non-overlapping, orthogonality, minimality, and consistency may be achieved.

Although we would like the design to satisfy all of the criteria, this is not possible because the criteria themselves are conflicting.

An example follows. As will be seen later, timestamp referencing, event extraction, and event time comparison are fundamental to valid-time selection. However, the (functionality-wise unnecessary) use of period comparison operators improve the readability. If we provide event operators only, the design satisfies the minimality and independence criteria, but not that of clarity. Using only event-based operators may result in confusing and erroneous predicates. Conflicts are resolved by retaining duplicating operators if they are used frequently or if their event-based equivalents are complicated. For example, the following two predicates are equivalent.

```
a OVERLAPS b
(END(a) > BEGIN(b)) and (END(b) > BEGIN(a))
```

The second predicate is hard to understand and `OVERLAPS` is a frequently used operator. In a case like this, the priority of clarity is higher than that of minimality and independence; `OVERLAPS` is included in the language.

4 Valid-time Selection in TSQL2

4.1 Overview

The design includes four types of valid-time timestamps, namely intervals, instants, periods, and elements. Intervals are directed, unanchored durations of time (e.g., 2 minutes), and while they cannot be uses as valid times, they may be used in expressions involving valid times. For details on intervals, see Chapter 21.

This section discusses three categories of operators related to valid-time selection, namely *extractors*, *constructors*, and *comparison operators*. Operators in the first category, e.g., `BEGIN`, create a new timestamp, e.g., a starting instant, by extraction from an argument timestamp, e.g., a period. To exemplify constructors, `INTERSECT` creates a period as the intersection of two overlapping periods. The operator `OVERLAPS` which tests whether two periods overlap illustrates comparison operators. As indicated, these categories of operators relate to instants as well as periods and elements.

The language does not include new clauses for timeslice and temporal predicates, both of which are commonly present in other temporal query languages.

Another characteristic is that valid-time selection and valid-time projection are considered orthogonal and therefore are completely separated in the design. Thus, the timestamps of results of queries are not defined by any valid-time selection operator. Temporal ordering is assumed to be the responsibility of aggregate functions and is thus not addressed here.

Table 6 is a summary of the language design. In that table, `event` denotes an argument of type `DATE`, `TIME`, or `TIMESTAMP`, `period` denotes an argument of period type, and `element` denote an argument of element type. The details are discussed in the next three sections.

4.2 Timestamp Referencing, Extraction and Construction

`VALID(`correlation name`)` is used for indicating timestamps of tuples. The alternative is an explicit reference such as `EMPLOYEE.PERIOD` where `EMPLOYEE` is a tuple variable and the postfix operator `PERIOD` returns the (period-valued) timestamp of an argument tuple. Another alternative is to overload correlation names to assume both a tuple and the timestamp of a tuple, depending on the context.

Several of the operators are adapted from Soo's proposal [Soo & Snodgrass 1992A]. Whereas TSQL and HSQL use a postfix notation for event extraction, TSQL2 employs a prefix, function-style notation for extraction. There are three reasons for this decision.

First, the prefix notation avoids confusing extraction with a reference to an attribute of a relation. For example, if `BEGIN` is an operator that extracts the first event of a timestamp, `EMPLOYEE.BEGIN` may be confused with a reference to an attribute `BEGIN` of the `EMPLOYEE` relation. In contrast, `BEGIN(EMPLOYEE)` does not have this problem of disallowing certain attribute names. Second, the prefix notation is more generally applicable than is the postfix notation which may not be used conveniently for operators that accept general temporal expressions as arguments. Third, other operators are prefix (or infix); thus, avoiding the postfix notation improves consistency.

The event extractors, `BEGIN` and `END`, return the first and last events, respectively, of an event, an period, or an element. The period extractors, `FIRST` and `LAST`, may be applied to timestamps of period or element type and return the first and last periods of the arguments, respectively.

The event constructors, `FIRST` and `LAST`, are new operators not provided by

Operation Type	Operators
timestamp referencing	`VALID(`correlation name`)`

event extraction	`BEGIN(event) BEGIN(period)`
	`BEGIN(element)`
	`END(event) END(period) END(element)`

period extraction	`FIRST(period) FIRST(element)`
	`LAST(period) LAST(element)`

event constructors	`FIRST(event, event)`
	`LAST(event, event)`

period constructors	`PERIOD(event, event)`
	`INTERSECT(period, period)`

element constructors	`INTERSECT(element, element)`
	`element + element`
	`element - element`
	May Also be Applied to
	Periods and Events

element comparison	`element PRECEDES element`
	`element = element`
	`element OVERLAPS element`
	`element CONTAINS element`

period comparison	`period PRECEDES period`
	`period = period`
	`period OVERLAPS period`
	`period MEETS period`
	`period CONTAINS period`

event time comparison	`event PRECEDES event`
	`event = event`
	`event OVERLAPS event`
	`event MEETS event`
	`event CONTAINS event`

mixed comparison	Comparison among elements, periods, and events
time slice clause	*None*
temporal ordering	*Use Aggregate Functions*
separate clause for valid-time selection	*No*
valid-time projection and selection	*Separated*

Table 6: Overview of valid-time selection

existing languages. Both of them take two events as arguments and return an event; `FIRST` returns the earliest event, and `LAST` returns the latest event. Following is an example showing the use of these two operators.

`PERIOD(FIRST(` e_1`,` e_2 `), LAST(` e_1`,` e_2`))`

There is hardly any clearer way to construct a period from two events whose order is not known.

The `PERIOD` function returns a period with two argument events as the delimiters. While the `INTERSECT` operator is not strictly necessary for reasons of functionality, it is still included because it is used frequently. This operator returns the period which is the intersection of two argument periods.

Three operators exist for constructing elements. The set of elements is closed under the binary operations of intersection, union, and difference. Thus, `INTERSECT`, `+`, and `-`, are included for constructing new elements. Since events and periods are special cases of elements, the former two may also appear as arguments.

4.3 Comparison Operators

It is essential that a temporal query language allows for the convenient comparison of timestamps.

A powerful set of comparison operators that satisfies the six design criteria is provided. For element comparison, `PRECEDES`, `=`, `OVERLAPS`, and `CONTAINS` are available; For period comparison `MEETS` is provided. Note that since events may be perceived as special cases of periods, events may occur as arguments where periods are allowed. Similarly, periods may occur where elements may occur. Table 7 gives the definition of these operators with the assumption that E_1 and E_2 are elements and I_1 and I_2 are periods. Operator `MEETS` has no natural generalization for elements. Observe that for events, operators `=`, `OVERLAPS`, and `CONTAINS` are equivalent.

Operator	Definition
E_1 `PRECEDES` E_2	`END(`E_1`)` is earlier than `BEGIN(`E_2`)`
E_1 `=` E_2	E_1 and E_2 are identical
E_1 `OVERLAPS` E_2	the intersection of E_1 and E_2 is not empty
E_1 `CONTAINS` E_2	each event in E_2 is contained in E_1
I_1 `MEETS` I_2	`END(`I_1`) PRECEDES BEGIN(`I_2`)` and there are no events between `END(`I_1`)` and `BEGIN(`I_2`)`

Table 7: Definition of comparison operators

This set of operators is complete in the sense that all possible relationships between two periods or two events are covered [1]. Indeed, a smaller set of operators could have been chosen had completeness, not user-friendliness, been the main concern. Table 8 show the equivalence of Allen's period comparison operators and TSQL2's operators.

Allen's Operators	TSQL2's Operators
`a before b`	`a PRECEDES b`
`a equal b`	`a = b`
`a overlaps b`	`a OVERLAPS b AND END(a) PRECEDES END(b)`
`a meets b`	`END(a) = BEGIN(b)`
`a during b`	`BEGIN(b) PRECEDES BEGIN(a)` `AND` `END(a) PRECEDES END(b)`
`a start b`	`BEGIN(a) = BEGIN(b)` `AND` `END(a) PRECEDES END(b)`
`a finish b`	`BEGIN(b) PRECEDES BEGIN(a)` `AND` `END(a) = END(b)`

Table 8: Comparison expressions in Allen's definition and TSQL2

According to Table 8, the first two pairs of operators are equivalent, with naming being the only difference. Allen's operator `overlaps` is a asymmetric. We have chosen the more common symmetric counterpart for TSQL2. Operator `meets` requires that the ending event of the first argument is identical to the starting event of the second argument. That is tested easily using event extraction and equality. The `MEETS` included here is harder to express with other operators and is expected to be at least as useful as `meets` in practice. Because the definitions of `during` and `CONTAINS` are slightly different (`a during b` means `a` started later than `b` and ended earlier than `b`), it is necessary to use two subexpressions to accomplish the definition of `during`. However, the operator `CONTAINS` as defined in TSQL2 is expected to be used more often than `during`. If a a "pure" `CONTAINS`, i.e., a `during`, is needed, the expression in the table provides the functionality.

Note that we do not provide the `ADJACENT` operator (found in TSQL) in TSQL2. This choice is made because the predicate, I_1 `ADJACENT` I_2, is equivalent to I_1 `MEETS` I_2 `OR` I_2 `MEETS` I_1, the semantics of which is clear.

The last two operators, `start` and `finish`, are expected to be used only sporadically in queries, so we do not provide equivalent operators for them. In total, we have attempted to strike a balance between minimality and clarity.

Below, two sample queries are expressed using TSQL2.

Q1. List all of the employees who worked during all of 1991.

```
SELECT Name
FROM Employee
WHERE VALID(Employee) CONTAINS
        PERIOD(DATE '01/01/1991', DATE '12/31/1991')
```

Q2. List all the employees who work in the company at some time when Tom is in the Toy department.

```
SELECT E1.Name
FROM Employee E1, Employee E2
WHERE E2.Name = "Tom" AND E2.Dept = "Toy"
    AND VALID(E1) OVERLAPS VALID(E2)
```

4.4 Additional Aspects

So far, we have accounted for most of Table 6, but a few design decisions still remain to be discussed.

We did not include a special timeslice clause and a separate clause for temporal selection. We also decided to separate temporal ordering from temporal selection. We now discuss the reasons for those decisions.

Most language proposals include a timeslice clause, but after a careful examination of timeslice, we have not included a special timeslice clause, for two reasons. First, a timeslice may be expressed in a clear way without using a special clause. For example, the timeslice in TSQL,

```
SELECT Name
FROM Employee
TIME-SLICE [1.1.1990, 12.31.1990]
```

is equivalent to

```
SELECT Name
FROM Employee
WHERE VALID(Employee) OVERLAPS
        PERIOD '01/01/1990 - 12/31/1990'
```

The meaning of this latter predicate is clear. Omitting a timeslice clause improves minimality without adversely affecting clarity. Second, the independence criterion for temporal predicates is violated if a timeslice clause is included.

Next, we present the considerations that led to not including a new clause for temporal selection. If we provide a new clause, e.g., `WHEN`, possible advantages include a increased syntactically clarity, separation of temporal predicates from non-temporal predicates, and a more structural language. Each of these may lead to increased clarity.

The advantages of not providing a new clause are less reserved words (minimality) and one clause for all predicates (consistency). Also, it is not clear what should be the boundary between a `WHEN` and a `WHERE` clause, i.e., what should go where. Because of user-defined time attributes, we cannot avoid temporal predicates in the `WHERE` clause totally. The advantage of separating temporal predicates from non-temporal predicates is then decreased. Further, it is still possible to write structured queries without a `WHEN` clause.

The language should allow the use of user-defined time attributes and valid times together in the same predicates. With a new clause, this combination of user-defined time with timestamps is not allowed. For example, adding a user-defined time attribute `PROJ-TIME` to the `EMPLOYEE` relation, the predicate

```
WHERE A.PROJ-TIME PRECEDES A
```

cannot be expressed using a combination of the `WHERE` clause and a new clause. Thus, it may be argued that adding a new clause actually decreases the expressive power of the language. Expressive power has the highest priority among the criteria, and it cannot be compromised for any reason.

We consider temporal ordering to be the responsibility of aggregate functions. In temporal ordering, a tuple is selected, not by testing it against a predicate, but by manipulating a group of tuples to get a correct order to obtain a desired tuples. Generally, whether a tuple is selected or not cannot be determined by testing the tuple against a predicate. An aggregate function takes a group of tuples, operates on all of the tuples, and returns a value. Temporal ordering functions fall into this category.

Aggregates are not covered in this chapter, so temporal ordering is not addressed.

The main reason to separate valid-time projection from temporal selection is added expressive power. When the two are mixed, it may be impossible or difficult to obtain adequate timestamps on result tuples. Separating valid-time selection and projection allows the two to be combined freely.

5 Valid-time Projection in TSQL2

We now discuss TSQL2's support for valid-time projection, which is specified in the optional <valid clause>. This clause consists of either `VALID` or `VALID INTERSECT` followed by a temporal expression.

5.1 Overview

Unlike some other proposed temporal query languages, we do not distinguish syntactically between instant and period projection. There are two reasons for this.

First, in standard SQL, no clauses have combinations of reserved words. Second, the compiler is capable of determining the correct type of a temporal expression and then define the correct timestamp type for the resulting relation.

However, we do support two different options in the valid clause: `VALID` and `VALID INTERSECT`. The reserved word `VALID` indicates a general valid-time projection, and all valid-time expressions returning periods or elements are allowed in the clause.

The alternative `VALID INTERSECT` indicates a restriction on the projection, namely that the resulting timestamp is the intersection of the time of the specified temporal expression and the timestamps of the relations (actually, correlation names) appearing in the `FROM` clause. If the given timestamps and temporal expression do not intersect, the resulting timestamp is empty, and hence that tuple doesn't participate further in the query. For example, the following two statements are equivalent.

```
VALID INTERSECT <temporal expression>
VALID INTERSECT(<temporal expression>,
                INTERSECT( <argument relations>))
```

From this, it is clear that `VALID INTERSECT` is subsumed by `VALID`. The reason for still having the alternative `VALID INTERSECT` is that it will be used very frequently in queries that should return tuples with valid times that do not extend beyond the valid times of the argument tuples. Put differently, using `VALID INTERSECT` ensures that queries do not return "manufactured" information that was not present in the database. Alternatively, using `VALID INTERSECT` restricts the possibilities for assigning timestamps to resulting tuples.

The `VALID` form is useful when new information, e.g., to be entered into the database, is computed from existing information. The following is a sample use of `VALID`.

Q3. Create a new department, Newtoy, from the original Toy department so that all of the employees currently in the Toy department will work in Newtoy one month from now.

```
INSERT INTO Newdept(Name, Dept)
    SELECT Name, 'Newtoy'
    VALID PERIOD(CURRENT_DATE + INTERVAL '1' MONTH,
                DATE 'forever')
    FROM Employee
    WHERE Dept = 'Toy'
        AND Employee OVERLAPS CURRENT_DATE)
```

In this example, we use the `VALID` form to specify the valid time for the new relation, `Newdept`. The `INTERVAL '1' MONTH` is an interval literal and indicates a duration of one month.

The timestamp in valid-time projection is specified as a valid-time expression which could be either period or element time.

The syntax and semantics of the valid-time expression in the valid clause is the same as the valid-time expressions specified in the `WHERE` clause. Thus, the period and element functions, as well as arithmetic operators, can all be used in valid-time projections.

The default value of the valid clause is the intersection of the timestamps of the argument relations, i.e., omitting `VALID` (`VALID INTERSECT`) is equivalent to "`VALID INTERSECT(`$relation_1$`,` $relation_2$`)`." If at least one of the argument relations is a snapshot relation, the default is also a snapshot relation. Table 9 shows the default value of the following (generic) query.

```
SELECT R1.A, R2.B
FROM R1, R2
WHERE ...
```

R1	R2	default value
valid time relation	valid time relation	`INTERSECT( R1, R2)`
valid time relation	snapshot relation	snapshot
snapshot relation	snapshot relation	snapshot

Table 9: Default values for the `VALID` clause

If the user wants a snapshot relation as a result of a query where the argument relations are valid-time relations, the reserved word `SNAPSHOT` is specified after `SELECT`. This use of `SNAPSHOT` is similar to the current use of `DISTINCT`. For example, the following query lists only the employees who have ever worked in the Toy department, with no timestamps in the resulting relation.

```
SELECT SNAPSHOT NAME
FROM EMP
WHERE DEPT = 'Toy'
```

In this section, we have introduced the valid clause; in the next section, we will motivate in more detail why the clause was included into the language.

5.2 Why a New Clause?

Originally, we wanted to add valid-time projection to the `SELECT` clause because valid-time projection is a kind of projection. This design would be consistent with the original SQL syntax.

However, it introduces a conflict with the earlier design decision of using the tuple-variable name for timestamp referencing. Adding valid-time projection to the

SELECT clause would mean that some attributes (i.e., the explicit, non-timestamp attributes) would be referenced via their names while the valid timestamp is referenced via the tuple-variable name.

Including the valid-time projection in the SELECT clause also implies that the valid time of a tuple is simply a regular attribute, not the underlying valid time of the entire tuple. On the other hand, the chosen design is consistent with the over-all special treatment of valid time in the query language.

Another disadvantage is that we need to add one reserved word for naming valid time in situations where a new relation is created. For example, **Q3** could be written as follows, where the reserved word VALIDTIMESTAMP is employed to indicate the implicit valid time attribute of the resulting relation.

```
INSERT INTO NEWDEPT(NAME, DEPT, VALIDTIMESTAMP)
SELECT NAME, 'NEWTOY',
    CURRENT_DATE + INTERVAL '1' MONTH
FROM EMPLOYEE
WHERE DEPT = 'Toy'
    AND OVERLAP(EMPLOYEE, CURRENT_DATE)
```

In summary, the major advantage of a new clause is clarity—where to define or read a valid-time projection is indicated clearly. In addition, having a new clause means that the SELECT clause is not complicated by a possibly lengthy valid-time projection. The presence of a new clause also emphasizes that the the valid time is not just another attribute, but is about the entire tuple.

One disadvantage is that a reserved word has been added. Another is that having a new clause for valid time may be claimed to be inconsistent with not having a new clause for valid-time selection.

Comparing the advantages and disadvantages of the two designs, we choose to include a separate clause for valid-time projection.

5.3 Discussion of Default Values

There are two obvious choices of default values for valid-time projection (the valid clause using the reserved word VALID). The first is the intersection of timestamps of the argument relations. The second is a snapshot relation, i.e., including no timestamp in the result relation. For example, the semantics of the following query is different under the two default values.

```
SELECT R1.A, R2.B
FROM R1, R2
WHERE R1.A = R2.C
```

If the default is intersection, the result is a valid-time relation, and the timestamp of each tuple is the intersection of timestamps of tuples from R1 and R2. However, if the default is a snapshot, the result is a snapshot relation with two attributes, A and B.

Table 10 is an overview of the differences between the two possible defaults. The first column indicates the types of two argument relations in a query. These can be either valid-time relations (VTR) or snapshot relations (SR). The last column indicates the type of timestamp accorded result tuples, with “snapshot” indicating not timestamps, “others” indicating some unspecified timestamp, and “R1 ∩ R2” indicating the intersection of the timestamps of the argument tuples. The two middle columns show what must be written in the valid clause to achieve certain kinds of timestamps given certain kinds of argument relations. The first column assumes that the default is “snapshot” and the second assumes that “intersection” is the default. In these two columns, “default” means that no clause is present, and “temporal expression” denotes an arbitrary valid-time expression.

Type of R1 and R2	Expression in the valid clause		Timestamp of the result
	Default is “snapshot”	Default is “R1 ∩ R2”	
R1, R2: VTR	intersect(R1,R2)	default	R1 ∩ R2
R1, R2: VTR	default	?	snapshot
R1, R2: VTR	temporal expression	temporal expression	others
R1, R2: SR	default	default	snapshot
R1:VTR, R2: SR	default	default	snapshot
R1:VTR, R2: SR	temporal expression	temporal expression	others

Table 10: Possible defaults for expressions in the valid clause

The table shows that there is not much difference between the two defaults. For the queries covered by rows three to six, either default works the same way. There is a difference only in the first two rows where the two argument relations are valid-time relations and where the desired result is a snapshot or a valid-time relation with intersection timestamps.

Because intersection of timestamps is believed to be used more often than producing snapshots, we choose the intersection of timestamps as the default value. The question mark in the second row indicates that some special mechanism is needed when a snapshot relation is to be produced from two valid-time relations.

Our solution is to add a new reserved word `SNAPSHOT` after `SELECT` to indicate that a snapshot relation is desired. The valid clause is simply left out. With the new reserved word, the syntax shows the resulting relation type explicitly. If we add the reserved word to the valid clause, the semantics are less clear. The reserved word is not a temporal expression and this is not consistent with the syntax of the valid clause. In SQL, the reserved word `DISTINCT` is also specified after `SELECT`. Thus, adding `SNAPSHOT` is consistent with SQL.

6 Summary

In a previous paper, we reviewed nine temporal query languages proposed in the past decade. These languages include five extensions to SQL, three extensions to Quel, and a procedural language. Because their underlying data models were not the same, it is not easy to compare the features in each language. However, all of the features, functionalities, new clauses, and reserved words of these languages were examined carefully when support for valid-time and projection in TSQL2 was designed. It has been a goal to build on the insights gained from the designs of previous temporal query languages.

Initially, six criteria, expressive power, consistency, clarity, minimality, orthogonality, and independence, were defined in order to guide the design.

We attempted to design simple but powerful language constructs with no unnecessary reserved words, clauses, and functions. Key features included a clean separation of valid-time selection and valid-time projection. The orthogonality of these constructs is reflected clearly in the design. No new clause was added for valid-time selection, mainly because of a desire to be able to mix valid time and user-defined time attributes in predicates. Defaults have been chosen carefully. While important, the notion of temporal ordering was not considered. The functionality of the temporal ordering is close to aggregate functions, and temporal ordering should be designed in that context.

A new clause for valid-time projection, with one reserved word, `VALID`, added. The former allows for assigning arbitrary timestamps to result tuples. The latter ensures that timestamp values of resulting tuples do not exceed the intersection of the timestamps of the argument tuples, i.e., it in not possible to manufacture information. We have thus allowed full flexibility (with `VALID`), but also have separated the “safe” queries (using `VALID INTERSECT`) from the potentially unsafe. Defaults have been chosen carefully. Most notably, a new reserved word `SNAPSHOT` is placed after `SELECT` to indicate that the result of a query should be a snapshot relation.

This chapter does not address the transfer of timestamp values to data structures of a programming language program execution via cursors. In SQL2, cursors can return values of explicit attributes; an extension is required to support the transfer of timestamp values, as discussed in Chapter 17.

References

[1] Allen, J. F. “Maintaining Knowledge about Temporal Intervals.” *Communications of the Association of Computing Machinery*, 26, No. 11, Nov. 1983, pp. 832–843.

[2] Ariav, G. “A Temporally Oriented Data Model.” *ACM Transactions on Database Systems*, 11, No. 4, Dec. 1986, pp. 499–527.

[3] Ben-Zvi, J. “The Time Relational Model.” PhD. Dissertation. Computer Science Department, UCLA, 1982.

[4] Chamberlain, D. D. “SEQUEL 2: A Unified Approach to Data Definition, Manipulation and Control..” *IBM Systems Journal*, 20, No. 6 (1976), pp. 560–575.

[5] 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.

[6] Gadia, S. K. “A Seamless Generic Extension of SQL for Querying Temporal Data.” Technical Report TR-92-02. Computer Science Department, Iowa State University. May 1992.

[7] Gadia, S. K. and J. H. Vaishnav. “A Query Language for a Homogeneous Temporal Database,” in *Proceedings of the ACM Symposium on Principles of Database Systems*. Mar. 1985, pp. 51–56.

[8] 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.

[9] Jones, S., P. Mason and R. Stamper. “LEGOL 2.0: A Relational Specification Language for Complex Rules.” *Information Systems*, 4, No. 4, Nov. 1979, pp. 293–305.

[10] Martin, N. G., S. B. Navathe and R. Ahmed. “Dealing with Temporal Schema Anomalies in History Databases,” in *Proceedings of the Conference on Very Large Databases*. Ed. P. Hammersley. Brighton, England: Sep. 1987, pp. 177–184.

[11] Navathe, S. B. and R. Ahmed. “A Temporal Relational Model and a Query Language.” UF-CIS Technical Report TR-85-16. Computer and Information Sciences Department, University of Florida. Apr. 1986.

[12] Navathe, S. B. and R. Ahmed. “TSQL-A Language Interface for History Databases,” in *Proceedings of the Conference on Temporal Aspects in Information Systems*. AFCET. France: May 1987, pp. 113–128.

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

[14] Sarda, N. L. “Extensions to SQL for Historical Databases.” *IEEE Transactions on Knowledge and Data Engineering*, 2, No. 2, June 1990, pp. 220–230.

[15] Sarda, N. L. “Algebra and Query Language for a Historical Data Model.” *The Computer Journal*, 33, No. 1, Feb. 1990, pp. 11–18.

[16] Snodgrass, R. T. 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.

[17] Snodgrass, R. T. (editor) “The TSQL2 Temporal Query Language.” Kluwer Academic Publishers, 1995.

[18] Tansel, A. U. and M. E. Arkun. “HQUEL, A Query Language for Historical Relational Databases,” in *Proceedings of the Third International Workshop on Statistical and Scientific Databases*. July 1986.